版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

仅供非商业用途或交流学习使用

Alibaba Group 阿里巴巴集团 | 阿里技术丛书系列

深入探索 Android 热修复技术原理

甘晓霖 廖斌斌 杨青 编著

多位大牛力荐 全方位系统介绍热修复原理，从阿里Sophix方案开发过程入手详细解读！

中国工信出版集团 電子工業出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY http://www.phei.com.cn

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

作者简介

甘晓霖，资深Android软件研发工程师。现就职于阿里巴巴手机淘宝，花名万壑。对Android系统底层机制与架构，尤其是Dalvik/Art虚拟机有着较为深入的研究。向Android系统源码提交过多个commit，并被Google采纳及并入主分支。阿里Android热修复方案Sophix的主要开发者。

廖斌斌，阿里巴巴高级无线开发工程师，花名悟二。拥有丰富的Android开发经验，同时热衷研究底层技术栈原理，目前主要在手机淘宝从事无线端SDK研发工作。

杨青，阿里巴巴高级专家，在无线互联网领域工作8年以上，拥有MTK、Symbian、PC、Windows Phone、Android、iOS等多个平台经验。曾参加国家自然科学基金项目，在国内外会议和期刊上发表论文3篇。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Alibaba Group 阿里巴巴集团 | 阿里技术丛书系列

深入探索 Android 热修复技术原理

甘晓霖 廖斌斌 杨青 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

内容简介

本书系统介绍了 Android 热修复的核心技术原理，结合 Sophix 热修复开发实践过程，从代码修复、资源修复、so库修复三大方向进行了详细的技术剖析与解读。

通过本书，读者会对 Android 热修复技术有更加深刻的认识，对于 Android 系统底层原理的理解和今后的开发工作都有很大帮助。通过阅读本书，读者可以初步实现一个较为完善的热修复框架。

本书适合对 Android 热修复技术感兴趣的技术人员阅读，也适合 Android进阶开发者参考。

本书著作权归阿里巴巴（中国）有限公司所有。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

深入探索Android热修复技术原理 / 甘晓霖，廖斌斌，杨青编著. —北京：电子工业出版社，2018.8
（阿里技术丛书系列）
ISBN 978-7-121-34389-6

Ⅰ. ①深… Ⅱ. ①甘… ②廖… ③杨… Ⅲ. ①移动终端－应用程序－程序设计
Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字（2018）第122912号

责任编辑：孙学瑛
印　　刷：北京捷迅佳彩印刷有限公司
装　　订：北京捷迅佳彩印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：720×1000　1/16　印张：14.75　字数：198千字
版　　次：2018 年8月第1版
印　　次：2018 年8月第1次印刷
定　　价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2014 年至今，手机淘宝引领了业界 Android 系统组件化和热修复技术的风潮，后来者 Instant App 或多或少地也受到了国内技术的影响。今天看到团队成员将热修复技术认真系统地整理成书，非常欣喜。在这本书中，既能看到对热修复技术发展历史系统深入的总结，又能看到国内程序员在 Android 系统级技术持续突破上做出的不懈努力，更可以看到国内程序员坚持打造优秀专业移动技术产品的雄心壮志！

吴志华（天施）

——手机淘宝基础平台部负责人，阿里巴巴资深技术专家

业内少有的深度讲解 Android 系统热修复技术的书籍，对于原理、代码讲解得非常清晰和深入，值得 Android 工程师研读。

倪生华（玄黎）

——手机淘宝基础架构团队负责人，阿里巴巴资深技术专家

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

深入探索 Android 热修复技术原理

应用热修复是一项略带神秘而又颇具争议的技术，但是它的确赋予了应用开发者“驾着飞机修引擎”的能力。本书从 Android 系统应用热修复技术的原理及代码实现、多种方案进行比较的角度，系统地阐述了 Android 平台的应用热修复技术。对 Android 系统应用热修复技术有好奇心的技术人员，这本专题书不容错过。

潘爱民

——计算机技术领域作家，阿里巴巴飞猪事业部首席架构师

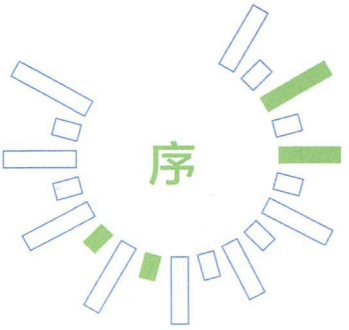
2015 年阿里无线在业界首次推出 Android 热修复技术 Dexposed，该技术为 Android 底层技术服务于业务痛点需求指明了一个崭新的方向，掀起了业界百花齐放的探索热潮。Sophix 的发布让我们再次看到了阿里无线在这个技术领域的自我迭代和锐意创新。这是一个技术改变格局的时代，同时也是一个能人辈出的时代！

冯森林

——安卓绿色联盟发起人，手机淘宝前架构师

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



随着无线互联网大行其道，经历了 Symbian、MTK、iOS、Android、Windows Phone 等各种系统，我们再一次见证了一个个操作系统的兴衰，这些兴衰有如下周期特征。

- 萌芽期：平台甫一出现，应用程序和开发者从 0 到 1 起步；
- 发展期：各领域的应用程序蓬勃发展，伴随黑产的兴起；
- 繁荣期：应用程序开始不断繁荣，甚至渴望突破用户态到管态的限制；
- 巅峰期：需求与安全领域的融合，对人无我有的追求，带来更多黑科技和动态性能力；
- 后期：应用和开发者增量逐步减少，进入平稳发展期或者衰退期。

尽管我们清楚地知道，不是所有平台都能完整经历一个兴衰周期，例如 MeeGo，但是可以发现操作系统的每一个发展阶段都缺少不了应用技术的身影，它在其中扮演了不可或缺的角色，而一个平台能真正进入巅峰期的标志是应用技术与安全领域的融合越来越深化。这一现象在各平台（从 Windows、Linux 到 Android、iOS）上也被多次验证，屡试不爽。以追求动态性修复能力的热修复技术为例，作为安全类技术的标志性衍生物，目前在 Android 平台上已

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

大行其道，这也标志着 Android 平台进入了巅峰期。

回首过去，Android 热修复技术在 2016 年如火如荼，无论是基于 Java/JVM 类加载机制的方案，还是利用 Android MultiDex 的方案，还是对 Android 组件偷梁换柱或进行 Hook、插件化等方案，无一不体现了应用技术的深化，也无一不蕴含了这样一个事实：开发者从获取快速流量转变为如何更好地维护流量。这意味着开发者的思维和研发模式发生了转变，应用技术的研发已进入游戏的下半程和深水区。我们也的确发现业界出现了各种热修复等动态化技术，它们的立足点或多或少都在强调修复之术的效果，而忽略了一个最基本的前提——产品化的程度，即对开发者接入或使用的各种微妙影响。为解决这一问题，我们从 Andfix 这样一个具有特色的技术开始，打造了一个友好的系统性热修复产品，这里既没有对系统组件的偷梁换柱，也没有运行期对补丁构建的大动干戈，还不需要定制编译工具和改变资源的排布，甚至不需要参与 APK 的构建。最终提出了一些原创性的发现，并兼具了及时性、无侵入性、高兼容性这三大原则，基本完成了当初做热修复产品的架构设计。

尽管如此，回首技术历史的长河，2016 年不一定会那么显著，但这一年中热修复等动态性技术层出不穷、直播 /AR/VR 方兴未艾，

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

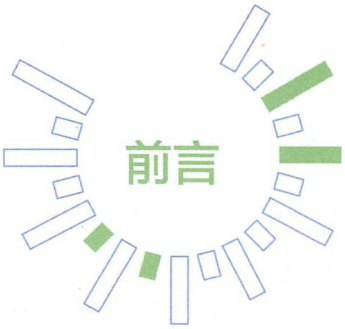
都反映出人们对应用技术的追求。而我们在 2016 年做出的种种尝试和决策，其结果和影响都将投射到历史的长河中，或许 2016 年会因为当时各种技术决策而成为一个承上启下的锚点，供未来审视，至少对于 Android 热修复领域会是这样，这或许是我们的收获。

读者通过阅读和研究本书的相关内容，能得到 Android 热修复的技术全貌，了解各种不同技术的特性和底层原理，还可以获得我们对于热修复技术的思考和解读，更深层次地了解我们对热修复技术本质的剖析，为 Android 热修复技术发展尽一份绵薄之力。

杨青

淘宝（中国）软件有限公司

仅供非商业用途或交流学习使用



热修复技术，可以看作 Android 平台发展成熟至一定阶段的必然产物。随着移动端业务复杂程度的增加，传统的版本更新流程显然无法满足业务和开发者的需求，热修复技术的推出在很大程度上改善了这一局面。热修复技术近年来的飞速发展，尤其是在 Instant Run 方案推出后，各种热修复技术百花齐放，国内大部分成熟的主流 App 都拥有自己的热更新技术，像手淘、支付宝、微信、QQ、饿了么、美团等。可以说，一个好的热修复技术，将为你的 App 助力百倍。对于每一个想在 Android 开发领域有所造诣的开发者，掌握热修复技术更是必备的素质。

虽然方案很多，但是深入系统地讲解热修复技术细节的书籍基本没有，市场上国内外的各种 Android 书籍大部分只是泛泛地囊括 Android 开发的基础知识，基本都没有包含热修复技术的章节，最多只是一笔带过。即使有很多开源的热修复方案，要自己硬啃代码还是要花费不少时间和精力。如果只有开源代码就足够的话，为什么还需要这么多技术书籍和文档呢？与其看一个晦涩难懂的项目，不如找一本专业翔实的书，这将会帮助你更系统、全面地理解这项技术涉及的难点与关键点。

出于回馈业界的考虑，我们把阿里移动热修复方案 Sophix 开

仅供非商业用途或交流学习使用

发过程中的技术细节进行了整理归纳，在 2017 年 6 月发布了《深入探索 Android 热修复技术原理》一书的免费电子版，与广大 Android 开发者进行了分享。

电子版发布以后得到了很好的反响，广大开发者下载如潮，很多读者希望我们能够推出纸质版，来进行更加方便的研读。基于这个呼吁，我们对电子版内容做了认真的整理和校对，对全书内容进行了全面的充实，特此推出这本《深入探索 Android 热修复技术原理》。

本书结合了这半年来我们的一些新的探索和思考，特别是对资源修复和冷启动修复技术进行了一些扩充。并新增了一章，全面介绍了其他热修复技术方案，其中特别邀请了 Andfix 的作者黎三平（董炼师）与 Amigo 的作者曹玉斌（夜沧），来专门为这两大优秀的热修复方案撰写相关章节，里面包含了两位作者分别在支付宝和饿了么的工程开发实践中，对于热修复技术的实际思考和研究，相信读者一定会获益不少。

本书结构

本书各章节是以热修复所涉及的各个技术点进行编排的，结构

分明、循序渐进。推荐以章节顺序进行阅读，当然如果对某些方面感兴趣，也可进行跳读。对于日常工作中遇到的问题，也可以通过阅读本书来寻求答案。

第 1 章　热修复技术介绍

热修复技术的演进与技术发展，Sophix 方案的简介。

第 2 章　热替换代码修复

详细解析了底层替换热修复的实现原理。

第 3 章　冷启动代码修复

对冷启动修复技术进行了深入的剖析。

第 4 章　资源热修复技术

资源修复的技术细节与思考。

第 5 章　so 库热修复技术

so 库修复的探索与实践。

第 6 章　其他优秀的热修复方案

介绍了 Android 平台上涌现的其他优秀热修复技术方案。

第 7 章　热修复技术的未来展望

对于热修复技术未来的畅想与期盼。

你将得到什么

读完本书，你将会对 Android 热修复技术有更加深刻的认识，不仅能提高工作效率，而且可加深对系统底层原理的理解，给日常的 Android 开发工作带来很大帮助。并且，目前热修复原理还是很多高级 Android 技术岗位的面试常客，对付它们你也将得心应手。你还可以通过本书的知识自己初步实现一个较为完善的热修复框架，当然，想直接使用可以访问阿里移动热修复 Sophix 官方地址[①]，马上就能够拥有安全可靠的全方位热修复功能。

致谢

Sophix 的推出与本书的发行是阿里巴巴许多开发者共同努力的成果，集团技术发展部对于本书的出版提供了极大的支持。

在这里首先要感谢团队领导所为，还有其他共同为 Sophix 的研发和推广做出贡献的悟二、查郁、泽胤、潇衍、荻朵，以及之前在百川项目里共同参与热修复项目开发的其他手机淘宝开发者。还要特别感谢阿里云事业部的同学们的合作，是你们提供了强有力的技术支持并不断开拓新市场。

① https://www.aliyun.com/product/hotfix

对于本人而言，我还要对前东家小米科技 MIUI 部门的各位朋友和前同事表示最诚挚的谢意。我个人的技术成长离不开你们曾经的帮助。

甘晓霖

淘宝（中国）软件有限公司

第 1 章
热修复技术介绍

热修复技术的演进与技术发展，
Sophix 方案的简介。

1.1 什么是热修复

对于广大的移动开发者而言，版本更新是最为寻常不过的事情了。然而，如果你发现刚发出去的包有紧急的 BUG 需要修复，那么就必须经历如图 1-1 所示的流程。

图 1-1 正常开发流程

这就是传统的版本更新流程，步骤十分烦琐，用户需要进入应用商店，下载完整的新版本安装包，花费大量时间等待安装完成，才能重新打开并使用修复完 BUG 的 App。如果这个 BUG 十分严重，有些用户可能就会对你失去耐心而直接卸载 App，你就再也没有修复 BUG 的机会了。

总体来说，传统流程存在这几大弊端。

- 重新发布版本代价太大；
- 用户下载安装成本太高；
- BUG 修复不及时，用户体验太差。

针对这种情况，许多开发者找到了比较合适的解决办法。其中一种办法是采用 Hybrid 方案。也就是把需要经常变更的业务逻辑以 H5 的方式独立出来。而这种方案，

需要传统的 Java 开发者学习前端语言，不仅增加了学习成本，而且还要对原有的逻辑进行合理的抽象和转换。但是，对于无法转为 H5 形式的代码中存在的 BUG 仍旧无法进行修复。

还有人会选择使用插件化方案来解决问题，像 Atlas 或者 DroidPlugin 方案。而这类方案，移植成本非常高，且不说要先学习整套插件化工具，对原有代码的改造也不是短时间内能够完成的。对于中小型 App 而言，此方案明显太过笨重。

于是，热修复技术应运而生了。

采用热修复技术，可以把更新补丁上传到云端，此时 App 就可以直接从云端下拉补丁直接应用生效。因此，更新流程就变为了这样，如图 1-2 所示。

图 1-2 热修复开发流程

可见，热修复的开发流程更加灵活，它有以下这几大优势。

- 无须重新发版，实时高效热修复；
- 用户无感知修复 BUG，无须下载新的应用，代价小；
- 修复 BUG 成功率高，把损失降到最低。

1.2 基本概念

我们知道，Android 的安装包是一个 APK 文件，APK 实际上是 zip 格式封装的，我们把它解压之后，可以看到里面的内容大致是这样的。

```
AndroidManifest.xml
META-INF/CERT.RSA
META-INF/CERT.SF
```

```
META-INF/MANIFEST.MF
classes.dex
resources.arsc
res/drawable/a.png
res/drawable/b.xml
res/layout/activity_main.xml
assets/c.png
lib/armeabi/libnative-lib.so
lib/x86/libnative-lib.so
```

AndroidManifest.xml 就不需要额外介绍了，只是这里的是二进制格式，并且合并了所有子项目的 AndroidManifest.xml 文件。

META-INF 下都是签名相关的文件，它可以校验 APK 中所有文件的合法性，从而识别唯一的开发者，不让其他人随意反编译修改你的 APK 文件。

classes.dex 应该是 APK 解压后文件中最重要的组成部分了，它是 Android 项目中所有 Java 代码最终编译后的形式。不管是开发者自己写的 Java 代码还是依赖的第三方库，都会由编译器经过 *.java → *.class → *.dex 的转换变为 Android 虚拟机可以识别运行的 dex 格式文件。这和传统的 JVM 应用（也就是桌面级的 Java 应用、服务端的 Java 应用，这类基于 Oracle 官方 Java 虚拟机的应用）是不同的，传统的 JVM 直接运行 .class 格式的文件，而 Android 由于是 Google 自己定制的 Dalvik/Art 虚拟机，所以运行的是对 .class 格式进行压缩合成的 .dex 格式文件。当然，如果是 Kotlin 或者其他 JVM 之上的语言，都会先转为 class 文件，然后再合并为 dex 文件，殊途同归。

resources.arsc 文件中包含了所有资源的 ID，以及它们具体的 ID 值和类型信息。平常开发中使用的 R.XX 这类的 ID，实际上都是一个由 32 位数字组成的资源 ID。它们有各种不同类型，要么是字符串，要么是数字，要么是资源路径。而如果是资源路径，则必然指代的是 res 文件中的资源文件。arsc 文件只是 ID 信息，而实际资源内容，像 XML 文件或者图片文件，都是放在 res 目录下的，并且在编译期间做过一些压缩处理。在程序运行过程中，会先从 arsc 文件中找到相应 ID 所对应的资源路径，然后再访问实际的 res 文件中路径所在的资源。

assets 中的文件也是资源，只是这些资源是不带资源 ID 的原始文件，因此，可以直接指定路径来访问这些资源。与 arsc 文件中的资源 ID 是完全不同的。

lib 文件下的就是 JNI 相关的 so 库了。这里的 armeabi 和 x86 下的 so 库，内容都是一样的，是所有 C/C++/Asm 代码编译后的最终二进制形式文件。只是在程序运行时，APK 会根据所安装 Android 设备的 CPU 的实际架构来选择具体加载哪个 so 库。

了解了 Android 系统的 APK 文件格式后，我们再来结合 App 的运行过程，看一下如何从本质上实现热修复功能。

首先，AndroidManifest 出现 BUG 是无法修复的，因为它是由系统进行解析的，系统会直接获取安装包里唯一的 AndroidManifest 文件，在解析过程中不会访问补丁包信息。因此如果想要增加四大组件，通常来说是不可以直接添加的。因为，我们需要在 AndroidManifest 里面加载新增组件时，通常的做法是预先在安装包的 AndroidManifest 里面埋入代理的组件，在每次新增组件时，进行偷梁换柱，通过预埋的代理组件实现与系统进程间的通信。

接下来，我们看看代码应该如何修复。由于所有 Java 代码最终都编译为 classes.dex 格式文件，因此任何的热修复方案，想要改变代码逻辑，都需要在补丁包里包含一个新逻辑的 dex 文件。然后在程序运行的时候加载这个 dex 文件，并且改变执行流，从执行原有安装包里的 classes.dex 文件引导到执行新 dex 文件。

而资源的修复，主要需要修改资源包的内容。正常情况下，资源包就是整个 APK 安装包，如果想要新增一个原有安装包里不存在的资源，就必须修改资源包的内容。所以，就必须想办法把原有的安装包替换为新资源包，或者把新的资源包插入程序的查找过程中。而有些资源，比如桌面图标、通知栏图标以及 RemoteView 之类的资源，是由系统直接解析安装包里的资源得到的，因此对于这类资源，任何热修复方案都无法进行资源替换和修复。

so 库的修复思路应该是最明确的。在 Android 系统中，所有 so 库都是由

System. load 进行加载的，因此只要找到办法在加载的时候优先加载补丁包的 so 库，而不是原有安装包的 so 库，就能够进行完整的底层代码替换了。

1.3 技术积淀

罗马不是一天建成的，阿里巴巴集团对 Android 系统热修复技术已经进行了多年的探索。

开始，是手机淘宝技术团队基于 Xposed 技术进行的改进，产生了针对 Android Dalvik 虚拟机运行时的 Java Method Hook 技术，即 Dexposed。但这个方案由于对底层 Dalvik 结构过于依赖，最终无法继续兼容 Android 5.0 版本以后的 ART 虚拟机，因此作罢。

后来支付宝技术团队提出了新的热修复方案 Andfix。Andfix 同样是一种底层结构替换的方案，也达到了运行时生效即时修复的效果，更重要的是，做到了 Dalvik 和 ART 环境的全版本兼容。阿里百川结合手机淘宝在实际工程中使用 Andfix 的经验，对相关业务逻辑解耦后，推出了阿里百川 Hotfix 方案，并在业内得到了良好的反响。

此时的百川 Hotfix 已经是一个很不错的产品了，对于基本的代码修复需求都可以满足，安全性和易用性都有提升。然而，它所依赖的基石，Andfix 本身，是有局限性的。且不说其底层固定结构的替换方案稳定性不好，单说其使用范围也存在着诸多限制，虽然可以通过改造代码绕过限制来达到相同的修复目的，但这种方式既不优雅也不方便。更大的问题是，Andfix 只提供了代码层的修复，对于资源层和 so 库层的修复都还未能实现。

而在 Android 平台上，业界除阿里系之外，比较著名的热修复方案还有：腾讯 QQ 空间的超级补丁技术、微信的 Tinker、饿了么的 Amigo、美团的 Robust 等。不过它们各自有自身的局限性，或者不够稳定，或者补丁过大，或者效率低下，或者使用起来过于烦琐，大部分方案技术上看起来似乎可行，但实际用户体验并不好。

终于，在 2017 年 6 月，阿里巴巴手机淘宝技术团队联合阿里云正式发布了新一代非侵入式 Android 热修复方案，即 Sophix。

Sophix 的横空出世，打破了各家热修复技术纷争的局面，在 Android 热修复的三大领域：代码修复、资源修复、so 库修复方面，以及方案的安全性和易用性方面，Sophix 都做到了业界领先。

表 1-1 从热修复最重要的几个维度，把 Sophix 和另外两个主流商业化热修复方案进行了对比。

表 1-1 Sophix 与 Tinker、Amigo 方案对比

方案对比	Sophix	Tinker	Amigo
dex 修复	**同时支持即时生效和冷启动修复**	冷启动修复	冷启动修复
资源更新	**差量包，不用合成**	差量包，需要合成	全量包，不用合成
so 库更新	**插桩实现，开发透明**	替换接口，开发不透明	插桩实现，开发透明
性能损耗	**低，仅冷启动情况下有些损耗**	高，有合成操作	低，全量替换
四大组件	**不能增加**	不能增加	能增加
生成补丁	**直接选择已经编译好的新旧包在本地生成**	编译新包时设置基线包	上传完整新包到服务端
补丁大小	**小**	小	大
接入成本	**傻瓜式接入**	复杂	一般
Android 版本	**全部支持**	**全部支持**	全部支持
安全机制	**加密传输及签名校验**	加密传输及签名校验	加密传输及签名校验
服务端支持	**支持服务端控制**	支持服务端控制	支持服务端控制

可以看到，Sophix 在各个指标上全面占优。而其中唯一不支持的地方就是四大组件的修复，这是因为如果要修复四大组件，必须在 AndroidManifest 里预先插入代理组件，并且尽可能声明所有权限，而这么做就会给原有的 App 添加很多臃肿的代码，对 App 运行流程的侵入性很强，所以，本着对开发者透明与代码极简的原则，

我们没有做这种多余的处理。

直接观察表 1-1 的话，其中有些技术细节可能还不太明朗，那么接下来，将从各个角度，深度解读 Sophix 的技术优势，以及它与同类技术的差别。

1.4 技术概览

Sophix 的诞生，起初是对原有的阿里百川 Hotfix 1.X 版本进行的升级衍进。

原有百川 Hotfix 服务端的整套请求控制流程，以及安全检查部分，是与热修复功能相对分离的，因此依旧保留了这部分的逻辑。

而原有的热修复方案，主要限制于 Andfix 本身，所以开始也是从突破原有修复的限制入手，希望能够基于原有的 Andfix 代码做一些必要的改进。然而最终发现，Andfix 自身的限制几乎是无法绕过的，在运行时原有类的结构已经固化在内存中，它的一些动态属性很难进行扩展。并且由于 Android 系统的碎片化，厂商的虚拟机底层结构都不是确定的，因此直接基于原有机制进行扩展的风险很大。

所以我们绕开了具体的技术实现细节，直接从修复的原理入手，对原有的代码修复技术进行深挖和改良。Sophix 的实现中，无处不体现着我们对易用性和优雅性的极致追求，在技术先进性与易用性上我们也达到了完美的平衡。

1.4.1 设计理念

Sophix 的核心设计理念，就是非侵入性。

Sophix 的打包过程不会侵入 APK 的构建流程中。我们所需要的只是已经生成完毕的新旧 APK，而至于 APK 是如何生成的则不需要关心，无论是 Android Studio 打包出来的、还是 Eclipse 打包出来的，或者是自定义的打包流程。在生成补丁的过程中既不会改变任何打包组件，也不插入任何 AOP 代码，我们极力做到了不添加任何

超出开发者预期的代码，以避免多余的热修复代码给开发者带来困扰。

在 Sophix 方案中，唯一需要的就是初始化和请求补丁的两行代码，甚至连 Application 入口类都不需要做任何修改，这样就给开发者带来了最大的透明度和自由度。我们甚至重新开发了打包工具，使得补丁工具操作图形界面化，这种所见即所得的补丁生成方式也是 Sophix 独家的。因此，Sophix 的接入成本也是目前市场上所有方案中最低的。

这种非侵入式热修复理念，是我们在设计过程中从用户使用角度进行了深入思考而提炼出来的。

这里的用户，指的自然是广大的开发者。对于开发者而言，热修复应该是一个与业务逻辑无关的 SDK 组件，在整个开发过程中感知不到它的存在。最理想的情况，就是开发者拿过来两个 APK，一个是已经安装在手机上的 APK，另一个是将要发布的 APK，直接通过工具，就可以根据这两个 APK 生成补丁，然后把补丁下发给已经安装的旧 App 上，就可以直接加载，使旧 App 重生为新的 App。而这个加载了补丁包的新 App，在功能和使用上，将会和直接安装新 APK 别无二致。

至于 Sophix 这个名字，读音是 ['sɔfiks]，是来源于 Sophic（明智的）+ Fix，一个更明智的热修复方案。

1.4.2 代码修复

代码修复有两大主要方案，一种是阿里系的底层替换方案，另一种是腾讯系的类加载方案。

这两类方案各有优缺点。

- 底层替换方案限制颇多，但时效性最好，加载轻快，立即见效。
- 类加载方案时效性差，需要重新冷启动才能见效，但修复范围广，限制少。

（1）底层替换方案

底层替换方案是在已经加载的类中直接替换原有方法，是在原有类的基础上进行修改的，因而无法实现对与原有类进行方法和字段的增减，因为这样将破坏原有类的结构。

一旦补丁类中出现了方法个数的增加或减少，就会导致这个类及整个 dex 的方法个数的变化。方法个数的变化伴随着方法索引的变化，这样在访问方法时就无法正常索引到正确的方法了。如果字段发生了增加或减少，和方法个数变化的情况一样，所有字段的索引都会发生变化。而且更严重的问题是，如果在程序运行中某个类突然增加了一个字段，那么对于已经产生的这个类的实例，它们还是原来的结构，这是无法改变的。而新方法使用到这些老的实例对象时，访问新增字段就会产生不可预期的后果。

这是这类方案的固有限制，而底层替换方案最为人诟病的地方在于底层替换的不稳定性。

传统的底层替换方式，不论是 Dexposed、Andfix 还是其他安全界的 Hook 方案，都是依赖直接修改虚拟机方法实体的具体字段实现的。例如，修改 Dalvik 方法的 JNI 函数指针、修改类或方法的访问权限等。这样就带来一个很严重的问题，由于 Android 系统是开源的，各个手机厂商都可以对代码进行改造，而 Andfix 里 ArtMethod 结构体的结构是根据公开的 Android 源码中的结构固定编写的。如果某个厂商对这个 ArtMethod 结构体的结构进行了修改，就和原有开源代码里的结构不一致，那么在这个修改过 AvtMethcd 结构体的设备上，通用性的替换机制就会出现问题。这便是不稳定的根源。

而我们也重新对代码的底层替换原理进行了深入思考，从如何克服其固有限制和兼容性入手，以一种更加优雅的替换思路，实现了即时生效的代码热修复方案。

我们实现的是一种不修改底层具体结构的替换方式，这种方式不仅解决了兼容性

问题，而且由于忽略了底层 ArtMethod 结构的差异，对于所有的 Android 版本都不再需要区分，代码量大大减少。即使以后的 Android 版本不断修改 ArtMethod 的结构，只要保证 ArtMethod 数结构体是以线性结构排列，就能直接适用于将来的 Android 8.0、9.0 等新版本，无需再针对新的系统版本进行适配。事实也证明确实如此，当我们拿到 Google 刚发布不久的 Android O(8.0) 开发者预览版系统时，Hotfix demo 直接就能顺利地加载补丁正常运行，我们并没有做任何适配工作，稳定性极好。

（2）类加载方案

类加载方案的原理是在 App 重新启动后让 Classloader 去加载新的类。因为在 App 启动到一半的时候，所有需要发生变更的类已经被加载过了，在 Android 系统上是无法对一个类进行卸载的。如果不重启，原有的类还在虚拟机中，就无法加载新类。因此，只有在下次重启的时候，在还没有运行到业务逻辑之前抢先加载补丁中的新类，这样后续访问这个类时，就会被解析为新类。从而达到热修复的目的。

再来看看腾讯系三大类加载方案的实现原理。QQ 空间方案会侵入打包流程，并且为 hack 添加一些无用的信息，实现起来很不优雅。而 Qfix 方案，需要获取底层虚拟机的函数，稳定性不够，并且有个比较严重的问题是无法新增 public 函数。

微信的 Tinker 方案是完整的全量 dex 文件加载，并且可以说是将补丁合成方案做到了极致，然而我们发现，精密的武器并非适用于所有战场。Tinker 的合成方案，是从 dex 的方法和指令维度进行全量合成，整个过程都是自己研发的。虽然可以很大程度上节省空间，但由于对 dex 内容的比较粒度过细，实现较为复杂，性能消耗比较严重。实际上，dex 的文件大小占整个 APK 的比例是比较低的，一个 App 里面的 dex 文件并不是主要部分，而占用空间大的主要还是资源文件。因此，Tinker 方案的时间与空面转换的性价比不高。

其实，dex 比较的最佳粒度，应该是类的维度。它既不像方法和指令维度那样的

仅供非商业用途或交流学习使用

细微，也不像 bsbiff 比较那般的粗糙。在类的维度上，可以达到时间和空间平衡的最佳效果。基于这个准则，我们另辟蹊径，实现了一种完全不同的全量 dex 替换方案。

我们采用的也是全量合成 dex 的技术，这个技术方案是从手机淘宝插件化框架 Atlas[①]汲取的。该方案会直接利用 Android 原有的类查找与合成机制，快速合成新的全量 dex 文件。这样，既不需要处理合成时方法数超过原有方法数的情况，也不会对 dex 的结构进行破坏性重构。

我们重新编排了包中 dex 文件的顺序。这样，在虚拟机查找类的时候，会优先找到 classes.dex 中的类，然后才是 classes2.dex、classes3.dex，也可以看作是 dex 文件级别的类插桩方案。这个方式十分巧妙，它对旧包与补丁包中的 classes.dex 顺序进行了打破与重组，最终使系统可以自然地识别到这个顺序，以实现类覆盖的目的。这将会大大减少合成补丁的开销。

（3）双剑合璧

既然底层替换方案和类加载方案各有其优点，把它们联合起来不是最好的选择吗？ Sophix 的代码修复体系正是同时涵盖了这两种方案。两种方案的结合，可以实现优势互补，完全兼顾的作用，可以灵活地根据实际情况自动切换。

这两种方案都进行了重大的改进，并且从补丁生成到应用的各个环节都进行了研究，使得二者能很好地整合在一起。在补丁生成阶段，补丁工具会根据实际代码变动情况进行自动选择，针对小修改，在底层替换方案限制范围内的，就直接采用底层替换修复，这样可以做到代码修复即时生效。而对于代码修改超出底层替换限制的，会使用类加载替换，这样虽然及时性没那么好，但总归可以达到热修复的目的。

另外，运行时阶段，Sophix 还会再次判断所运行的机型是否支持热修复，这样即使补丁支持热修复，但由于机型底层虚拟机构造不支持热修复，还是会走类加载修复方案，从而达到最好的兼容性。

① https://github.com/alibaba/atlas

仅供非商业用途或交流学习使用

1.4.3 资源修复

目前市场上的很多资源热修复方案基本上都是参考了 Instant Run 的实现。实际上，Instant Run 的推出正是推动这次热修复浪潮的主因，各家的热修复方案，在代码、资源等方面的实现，很大程度上参考了 Instant Run 的代码，而资源修复方案则被参考的最多。

简单说，Instant Run 中的资源热修复分为两步。

1. 构造一个新的 AssetManager，并通过反射调用 addAssetPath 函数，然后把完整的新资源包加载到 AssetManager 中。这样就得到了一个含有所有新资源的 AssetManager。
2. 找到所有之前引用到原有 AssetManager 的地方，通过反射，把引用处替换为新的 AssetManager。

我们发现，其实大量代码都是在处理兼容性问题和搜索到 AssetManager 所有的引用处，真正的替换逻辑其实很简单。

我们的方案没有直接参考 Instant Run 的技术，而是另辟蹊径，构造了一个 package id 为 0x66 的资源包，这个包里只包含需要改变的资源项，然后直接在原有 AssetManager 中通过 addAssetPath 函数添加这个包就可以了。由于补丁包的 package id 为 0x66，不与目前已经加载的地址为 0x7f 的包冲突，因此直接加载到已有的 AssetManager 中就可以直接使用了。补丁包里面的资源，只包含原有包里面没有的新增资源，以及原有内容发生了改变的资源。并且，我们采用了更加优雅的替换方式，直接在原有的 AssetManager 对象上进行解析和重构，这样所有原有对 AssetManager 对象的引用是没有发生改变的，所以就不需要像 Instant Run 那样进行烦琐的修改了。

可以说，我们的资源修复方案，性能超过了 Google 官方的 Instant Run 方案。整个资源替换的方案优势如下。

- 不修改 AssetManager 的引用处，替换资源更快更完全（对比 Instanat

仅供非商业用途或交流学习使用

Run 以及所有 CopyCat 的实现）。

- 不必下发完整包，补丁包中只包含变动的资源（对比 Instanat Run、Amigo 等方式的实现）。
- 不需要在运行时合成完整包，不占用运行时计算和内存资源（对比 Tinker 的实现）。

所以，我们不要被所谓的“官方实现”束缚住手脚，其实 Instant Run 的开发团队和 Android Framework 的开发团队并不是同一个团队，他们对于 Android 系统机制的理解也未必十分深入。只要你认真研读系统代码，实现一个比官方更好的方案绝非难事。所以，要想实现技术方案的突破，首先就需要破除所谓“权威”的观念。

1.4.4 so 库修复

so 库的修复本质上是对 native 方法的修复和替换。

我们采用的是类似类修复反射注入方式。把补丁 so 库的路径插入到 nativeLibraryDirectories 数组的最前面，就能够达到加载 so 库的时候是补丁 so 库的目录从而修复 BUG 的目的。

采用这种方案，完全由 Sophix 在启动期间反射注入补丁中的 so 库。方案对开发者依然是透明的，不用像某些其他方案需要手动替换系统的 System.load 来实现替换功能。

1.5 本章小结

本章介绍了热修复技术的主要使用场景和为业内带来的变化。详细说明了阿里巴巴集团推出的热修复解决方案 Sophix 的由来，同时与其他各大主流方案进行了比较。并且概括了热修复技术所涉及的三大方向：代码修复、资源修复、so 修复的实现方式，后续章节根据这三大方向展开讲解。

第 2 章
热替换代码修复

详细解析了底层替换热修复的实现原理。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.1 底层热替换原理

在各种 Android 热修复方案中，Andfix 的即时生效特征令人印象深刻，它并不需要重新启动 App，而是在加载补丁后直接对方法进行替换就可以完成修复，然而它的使用限制也遭受到更多的质疑。

2.1.1 Andfix 回顾

我们先来看一下，为何唯独 Andfix 能够做到即时生效呢?

原因是这样的，在 App 启动到一半的时候，所有需要发生变更的分类已经被加载过了，在 Android 系统中是无法对一个分类进行卸载的。而腾讯系的方案是让 Classloader 去加载新的类，如果不重启 App，原有的类还在虚拟机中，就无法加载新类。因此，只有在下次 App 重启的时候，在还没运行到业务逻辑之前抢先加载补丁中的新类，这样在后续访问这个类时，就会解析为新的类。从而达到热修复的目的。

Andfix 采用的方法是直接在已经加载的类中的 native 层替换掉原有方法，是在原有类的基础上进行修改的。我们这就来看一下 Andfix 的具体实现。

其核心在于 replaceMethod 函数。

文件：AndFix/src/com/alipay/euler/andfix/AndFix.java。

```
private static native void replaceMethod(Method src, Method dest);
```

仅供非商业用途或交流学习使用

这是一个 native 方法，它的参数是在 Java 层通过反射机制得到的 Method 对象所对应的 jobject。src 对应的是需要被替换的原有方法。而 dest 对应的就是新方法，新方法存在于补丁包的新类中，也就是补丁方法。

文件：AndFix/jni/andfix.cpp。

```
static void replaceMethod(JNIEnv* env, jclass clazz, jobject src,
        jobject dest) {
    if (isArt) {
        art_replaceMethod(env, src, dest);
    } else {
        dalvik_replaceMethod(env, src, dest);
    }
}
```

Android 的 Java 运行环境，在 4.4 以下版本用的是 Dalvik 虚拟机，而在 4.4 以上版本用的是 Art 虚拟机。

文件：AndFix/jni/art/art_method_replace.cpp。

```
extern void __attribute__ ((visibility ("hidden"))) art_replaceMethod(
        JNIEnv* env, jobject src, jobject dest) {
    if (apilevel > 23) {
        replace_7_0(env, src, dest);
    } else if (apilevel > 22) {
        replace_6_0(env, src, dest);
    } else if (apilevel > 21) {
        replace_5_1(env, src, dest);
    } else if (apilevel > 19) {
        replace_5_0(env, src, dest);
    }else{
        replace_4_4(env, src, dest);
    }
}
```

我们以 Art 虚拟机为例，对于不同 Android 版本的 Art 虚拟机，底层 Java 对象的数据结构是不同的，因而会进一步区分出不同的替换函数，这里我们以 Android 6.0 版本为例，对应的就是 `replace_6_0`。

仅供非商业用途或交流学习使用

文件：AndFix/jni/art/art_method_replace_6_0.cpp。

```
void replace_6_0(JNIEnv* env, jobject src, jobject dest) {

    // %% 通过 Method 对象得到底层 Java 函数对应 ArtMethod 的真实地址
    art::mirror::ArtMethod* smeth =
            (art::mirror::ArtMethod*) env->FromReflectedMethod(src);

    art::mirror::ArtMethod* dmeth =
            (art::mirror::ArtMethod*) env->FromReflectedMethod(dest);

    // %% 把旧函数的所有成员变量都替换为新函数的
    smeth->declaring_class_ = dmeth->declaring_class_;
    smeth->dex_cache_resolved_methods_ = dmeth->dex_cache_resolved_methods_;
    smeth->dex_cache_resolved_types_ = dmeth->dex_cache_resolved_types_;
    smeth->access_flags_ = dmeth->access_flags_;
    smeth->dex_code_item_offset_ = dmeth->dex_code_item_offset_;
    smeth->dex_method_index_ = dmeth->dex_method_index_;
    smeth->method_index_ = dmeth->method_index_;

    smeth->ptr_sized_fields_.entry_point_from_interpreter_ =
    dmeth->ptr_sized_fields_.entry_point_from_interpreter_;

    smeth->ptr_sized_fields_.entry_point_from_jni_ =
    dmeth->ptr_sized_fields_.entry_point_from_jni_;
    smeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_ =
    dmeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_;

    LOGD("replace_6_0: %d , %d",
         smeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_,
         dmeth->ptr_sized_fields_.entry_point_from_quick_compiled_code_);
}
```

每一个 Java 方法在 Art 虚拟机中都对应着一个 ArtMethod，ArtMethod 记录了这个 Java 方法的所有信息，包括所属类、访问权限、代码执行地址等。

通过 `env->FromReflectedMethod`，可以由 Method 对象得到这个方法所对应的 ArtMethod 的真正起始地址，然后就可以把它强制转化为 ArtMethod 指针，从而对其包含的所有成员进行修改。

这样全部替换完之后就完成了热修复功能。以后调用方法时就会直接运行到新方法中实现。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.1.2 虚拟机调用方法的原理

为什么这样替换完就可以实现热修复呢？这需要从虚拟机调用方法的原理说起。

在 Android 6.0 版本中，Art 虚拟机中 ArtMethod 的结构是如下：

文件：art/runtime/art_method.h。

```
class ArtMethod FINAL {

 protected:
  // Field order required by test "ValidateFieldOrderOfJavaCppUnionClasses".
  // The class we are a part of.
  GcRoot<mirror::Class> declaring_class_;

  // Short cuts to declaring_class_->dex_cache_ member for fast compiled code access.
  GcRoot<mirror::PointerArray> dex_cache_resolved_methods_;

  // Short cuts to declaring_class_->dex_cache_ member for fast compiled code access.
  GcRoot<mirror::ObjectArray<mirror::Class>> dex_cache_resolved_types_;

  // Access flags; low 16 bits are defined by spec.
  uint32_t access_flags_;

  /* Dex file fields. The defining dex file is available via declaring_
     class_->dex_cache_ */

  // Offset to the CodeItem.
  uint32_t dex_code_item_offset_;

  // Index into method_ids of the dex file associated with this method.
  uint32_t dex_method_index_;

  /* End of dex file fields. */

  /* Entry within a dispatch table for this method. For static/direct methods
     the index is into */
  /* the declaringClass.directMethods, for virtual methods the vtable and for
     interface methods the */
  // ifTable.
  uint32_t method_index_;

  // Fake padding field gets inserted here.

  // Must be the last fields in the method.
  // PACKED(4) is necessary for the correctness of
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
  // RoundUp(OFFSETOF_MEMBER(ArtMethod, ptr_sized_fields_), pointer_size).
  struct PACKED(4) PtrSizedFields {
    /* Method dispatch from the interpreter invokes this pointer which may
       cause a bridge into */
    // compiled code.
    void* entry_point_from_interpreter_;

    /* Pointer to JNI function registered to this method, or a function to
       resolve the JNI function. */
    void* entry_point_from_jni_;

    /* Method dispatch from quick compiled code invokes this pointer which
       may cause bridging into */
    // the interpreter.
    void* entry_point_from_quick_compiled_code_;
  } ptr_sized_fields_;

}
```

其中最重要的字段就是 entry_point_from_interprete_ 和 entry_point_from_quick_compiled_code_ 了，从名字可以看出来，它们就是方法的执行入口。我们知道，Java 代码在 Android 中会被编译为 Dex Code。

Art 虚拟机中可以采用解释模式或者 AOT 机器码模式执行 Dex Code。

解释模式，就是取出 Dex Code，逐条解释执行。如果方法的调用者是以解释模式运行的，在调用这个方法时，就会获取这个方法的 entry_point_from_interpreter_，然后跳转执行。

而如果是采用 AOT 的方式，就会预先编译好 Dex Code 对应的机器码，然后在运行期直接执行机器码，不需要逐条解释执行 Dex Code。如果方法的调用者是以 AOT 机器码方式执行的，在调用这个方法时，就是跳转到 entry_point_from_quick_compiled_code_ 中执行。

那是不是只需要替换这几个 entry_point_* 入口地址就能够实现方法替换了呢?

并没有这么简单，因为不论是解释模式还是 AOT 机器码模式，在运行期间还会需要调用 ArtMethod 中的其他成员字段。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

就以 AOT 机器码模式为例，虽然 Dex Code 已被编译成了机器码。但是机器码并非可以脱离虚拟机而单独运行，以下面这段简单的代码为例。

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    ... ...
```

编译为 AOT 机器码后，是这样的：

```
  7: void com.patch.demo.MainActivity.onCreate(android.os.Bundle) (dex_
method_idx=20639)
    DEX CODE:
      0x0000: 6f20 4600 1000              | invoke-super {v0, v1}, void
android.App.Activity.onCreate(android.os.Bundle) // method@70
      0x0003: 0e00                        | return-void

    CODE: (code_offset=0x006fdbac size_offset=0x006fdba8 size=96)
      ... ...
      0x006fdbe0: f94003e0 ldr x0, [sp]
      ;x0 = MainActivity.onCreate 对应的 ArtMethod 指针
      0x006fdbe4: b9400400 ldr w0, [x0, #4]
      ;w0 = [x0 + 4] = dex_cache_ resolved_methods_ 字段
      0x006fdbe8: f9412000 ldr x0, [x0, #576]
      ;x0 = [x0 + 576];dex_cache_resolved_methods_ 数组的第 72(=576/8) 个元素，
      ; 即对应 Activity.onCreate 的 ArtMethod 指针

      0x006fdbec: f940181e ldr lr, [x0, #48]
      ;lr = [x0 + 48] = Activity.onCreate 的 ArtMethod 成员的执行入口点
      ; 即 entry_point_from_quick_compiled_code_
      0x006fdbf0: d63f03c0 blr lr; 调用 Activity.onCreate
```

这里面去掉了一些校验之类的无关代码，可以看到，在调用一个方法时，获取了 ArtMethod 中的 dex_cache_resolved_methods_ 文件，这是一个存放 ArtMethod* 的指针数组，通过它就可以访问到这个 Method 所在 Dex 中所有的 Method 所对应的 ArtMethod*。

Activity.onCreate 的方法索引是 70，由于是 64 位系统，因此每个指针的大小为 8KB，又由于 ArtMethod* 元素是从这个数组的第 0×2 个位置开始存放的，因此偏移量为 (70 + 2) × 8 = 576 的位置正是 Activity.onCreate 的 ArtMethod 指针。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

这是一个比较简单的例子，而在实际代码中，有许多更为复杂的调用情况。很多情况下还需要调用 dex_code_item_offset_ 等字段。由此可以看出，AOT 机器码的执行过程，还是会有对虚拟机以及 ArtMethod 成员字段的依赖。

因此，当把一个旧方法的所有成员字段都换为新方法的成员字段后，执行时所有数据就可以保持和新方法的数据一致。这样在所有执行到旧方法的地方，会获取新方法的执行入口、所属类型、方法索引号以及所属 dex 信息，然后像调用旧方法一样执行新方法的逻辑。

2.1.3 兼容性问题的根源

然而，目前市场上几乎所有的 native 替换方案，比如 Andfix 和其他安全界的 Hook 方案，ArtMethod 结构体的结构都是固定的，这会带来巨大的兼容性问题。

从刚才的分析可以看到，虽然 Andfix 是把底层结构强转为 art::mirror::ArtMethod，但这里的 art::mirror::ArtMethod 并非等同于 App 运行时所在设备虚拟机底层的 art::mirror::ArtMethod，而是 Andfix 自己构造的 art::mirror::ArtMethod。

文件：AndFix/jni/art/art_6_0.h。

```
class ArtMethod {
public:

    // Field order required by test "ValidateFieldOrderOfJavaCppUnionClasses".
    // The class we are a part of.
    uint32_t declaring_class_;
    /* Short cuts to declaring_class_->dex_cache_ member for fast compiled
       code access. */
    uint32_t dex_cache_resolved_methods_;
    /* Short cuts to declaring_class_->dex_cache_ member for fast compiled
       code access. */
    uint32_t dex_cache_resolved_types_;
    // Access flags; low 16 bits are defined by spec.
    uint32_t access_flags_;
    /* Dex file fields. The defining dex file is available via declaring_
class_->dex_cache_ */
    // Offset to the CodeItem.
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    uint32_t dex_code_item_offset_;
    // Index into method_ids of the dex file associated with this method.
    uint32_t dex_method_index_;
    /* End of dex file fields. */
    /* Entry within a dispatch table for this method. For static/direct
       methods the index is into */
    /* the declaringClass.directMethods, for virtual methods the vtable and
       for interface methods the */
    // ifTable.
    uint32_t method_index_;
    // Fake padding field gets inserted here.
    // Must be the last fields in the method.
    // PACKED(4) is necessary for the correctness of
    // RoundUp(OFFSETOF_MEMBER(ArtMethod, ptr_sized_fields_), pointer_size).
    struct PtrSizedFields {
        /* Method dispatch from the interpreter invokes this pointer which
           may cause a bridge into */
        // compiled code.
        void* entry_point_from_interpreter_;
        /* Pointer to JNI function registered to this method, or a function
           to resolve the JNI function. */
        void* entry_point_from_jni_;
        /* Method dispatch from quick compiled code invokes this pointer
           which may cause bridging into */
        // the interpreter.
        void* entry_point_from_quick_compiled_code_;
    } ptr_sized_fields_;
};
```

我们再来回顾一下 Android 开源代码里面 Art 虚拟机里的 ArtMethod:

文件：art/runtime/art_method.h。

```
class ArtMethod FINAL {
 ... ...

 protected:
  // Field order required by test "ValidateFieldOrderOfJavaCppUnionClasses".
  // The class we are a part of.
  GcRoot<mirror::Class> declaring_class_;

  // Short cuts to declaring_class_->dex_cache_ member for fast compiled code access.
  GcRoot<mirror::PointerArray> dex_cache_resolved_methods_;

  // Short cuts to declaring_class_->dex_cache_ member for fast compiled code access.
  GcRoot<mirror::ObjectArray<mirror::Class>> dex_cache_resolved_types_;

  // Access flags; low 16 bits are defined by spec.
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
    uint32_t access_flags_;

    /* Dex file fields. The defining dex file is available via declaring_
       class_->dex_cache_ */

    // Offset to the CodeItem.
    uint32_t dex_code_item_offset_;

    // Index into method_ids of the dex file associated with this method.
    uint32_t dex_method_index_;

    /* End of dex file fields. */

    /* Entry within a dispatch table for this method. For static/direct methods
       the index is into */
    /* the declaringClass.directMethods, for virtual methods the vtable and for
       interface methods the */
    // ifTable.
    uint32_t method_index_;

    // Fake padding field gets inserted here.

    // Must be the last fields in the method.
    // PACKED(4) is necessary for the correctness of
    // RoundUp(OFFSETOF_MEMBER(ArtMethod, ptr_sized_fields_), pointer_size).
    struct PACKED(4) PtrSizedFields {
      /* Method dispatch from the interpreter invokes this pointer which may
         cause a bridge into */
      // compiled code.
      void* entry_point_from_interpreter_;

      /* Pointer to JNI function registered to this method, or a function to
         resolve the JNI function. */
      void* entry_point_from_jni_;

      /* Method dispatch from quick compiled code invokes this pointer which
         may cause bridging into */
      // the interpreter.
      void* entry_point_from_quick_compiled_code_;
    } ptr_sized_fields_;

... ...
}
```

可以看到，ArtMethod 结构中的各个成员的大小是和 AOSP 开源代码里完全一致的。这是由于 Android 源码是开源的，Andfix 里面的这个 ArtMethod 自然是遵照 Android 虚拟机 Art 源码里面的 ArtMethod 构建的。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

但是，由于 Android 源码是开源的，各个手机厂商都可以对代码进行改造，而 Andfix 里 ArtMethod 的结构是根据公开的 Android 源码中的结构编写的。如果某个厂商对这个 ArtMethod 结构体进行了修改，就和原有开源代码里的结构不一致，那么在这个修改过 ArtMethod 结构体的设备上，替换机制就会出问题。

比如，在 Andfix 替换 `declaring_class_` 的地方，

```
smeth->declaring_class_ = dmeth->declaring_class_;
```

由于 `declaring_class_` 是 Andfix 里 ArtMethod 的第一个成员，因此它和以下这行代码等价：

```
*(uint32_t*) (smeth + 0) = *(uint32_t*) (dmeth + 0)
```

如果某手机厂商在 ArtMethod 结构体中的 `declaring_class_` 前面添加了一个字段 `additional_`，那么，additional_ 就成为了 ArtMethod 的第一个成员，所以 smeth + 0 这个位置在这台设备上实际就变成了 `additional_`，而不再是 `declaring_class_` 字段。所以这行代码的真正含义就变成了：

```
smeth->additional_ = dmeth->additional_;
```

这样就和原有替换 `declaring_class_` 的逻辑不一致，从而无法正常执行热修复逻辑。

这也正是 Andfix 不支持所有机型的原因，很大的可能，是因为这些机型修改了底层的虚拟机结构。

2.2 突破底层差异的方法

2.2.1 突破底层结构差异

知道了 native 替换方式兼容性问题的原因，我们是否有办法寻求一种新的方式，

仅供非商业用途或交流学习使用

不依赖 ROM 底层方法结构的实现而达到替换效果呢?

我们发现，这样的 native 层面替换思路，其实就是替换 ArtMethod 的所有成员。那么，并不需要构造出 ArtMethod 具体的各个成员字段，只要把 ArtMethod 作为整体进行替换，这样不就可以了吗?

也就是把原有的逐一替换，如图 2-1 所示。

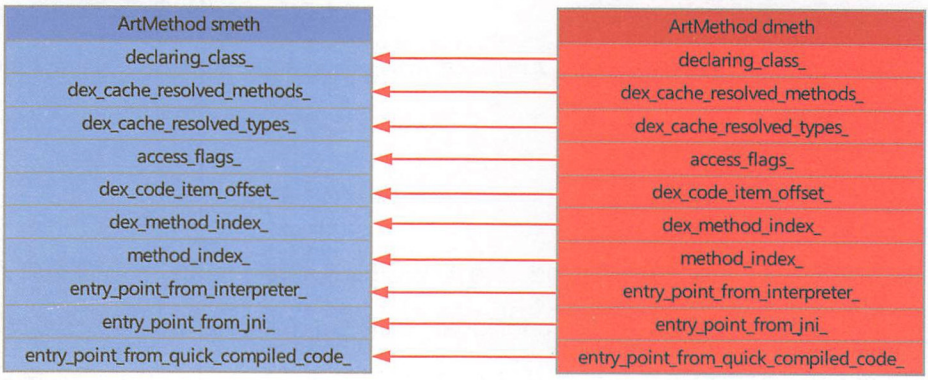


图 2-1　逐一替换方案

变成整体替换替换方案，如图 2-2 所示。

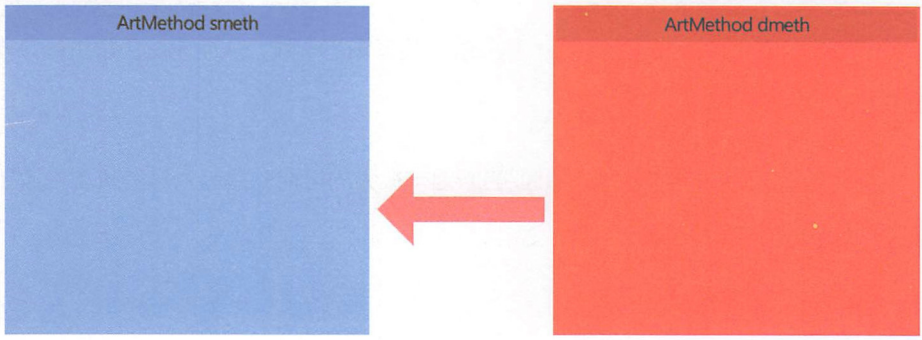


图 2-2　整体替换方案

因此 Andfix 这一系列烦琐的替换：

```
// %% 把旧函数的所有成员变量都替换为新函数的
smeth->declaring_class_ = dmeth->declaring_class_;
```


仅供非商业用途或交流学习使用

```
smeth->dex_cache_resolved_methods_ = dmeth->dex_cache_resolved_methods_;
smeth->dex_cache_resolved_types_ = dmeth->dex_cache_resolved_types_;
smeth->access_flags_ = dmeth->access_flags_;
smeth->dex_code_item_offset_ = dmeth->dex_code_item_offset_;
smeth->dex_method_index_ = dmeth->dex_method_index_;
smeth->method_index_ = dmeth->method_index_;
... ...
```

其实可以缩写为：

```
memcpy(smeth, dmeth, sizeof(ArtMethod));
```

就是这样的一句代码就能取代上面的一堆代码，这正是我们深入理解替换机制的本质之后研发出的新替换方案。

刚才提到过，不同的手机厂商都可以对底层的 ArtMethod 结构进行任意修改，但即使他们把 ArtMethod 改得面目全非，只要像这样把 ArtMethod 整个结构体完整替换，就能够把所有旧方法成员自动对应的换成新方法的成员。

但这其中最关键的地方，在于 sizeof(ArtMethod) 的计算结果，如果计算结果有偏差，导致部分成员没有被替换，或者替换区域超出了边界，都会导致严重的问题。

对于 ROM 开发者而言，是在 Art 源代码中开发，所以一个简单的 `sizeof(ArtMethod)` 就行了，因为这是在编译期就可以决定的。

但对于上层开发者，App 会被下发给各式各样的 Android 设备，所以需要在运行时动态地获取 App 所运行设备中的底层 ArtMethod 大小的，这就没那么简单了。

想要忽略 ArtMethod 的具体结构成员直接获取其 size 的精确值，还是需要从虚拟机的源码入手，**从底层的数据结构及排列特点探寻答案**。

在 Art 中，初始化一个类的时候会给这个类的所有方法分配内存空间，我们可以看到这个分配内存空间的地方：

仅供非商业用途或交流学习使用

文件：android-6.0.1_r62/art/runtime/class_linker.cc。

```
void ClassLinker::LoadClassMembers(Thread* self, const DexFile& dex_file,
                                   const uint8_t* class_data,
                                   Handle<mirror::Class> klass,
                                   const OatFile::OatClass* oat_class) {
    ... ...

    ArtMethod* const direct_methods = (it.NumDirectMethods() != 0)
        ? AllocArtMethodArray(self, it.NumDirectMethods())
        : nullptr;
    ArtMethod* const virtual_methods = (it.NumVirtualMethods() != 0)
        ? AllocArtMethodArray(self, it.NumVirtualMethods())
        : nullptr;

    ... ...
```

类的方法有 direct 方法和 virtual 方法。direct 方法包含 static 方法和所有不可继承的对象方法。而 virtual 方法包含所有可以继承的对象方法。

AllocArtMethodArray 函数分配了它们的方法所在区域。

文件：android-6.0.1_r62/art/runtime/class_linker.cc。

```
ArtMethod* ClassLinker::AllocArtMethodArray(Thread* self, size_t length) {
  const size_t method_size = ArtMethod::ObjectSize(image_pointer_size_);
  uintptr_t ptr = reinterpret_cast<uintptr_t>(
      Runtime::Current()->GetLinearAlloc()->Alloc(self, method_size * length));
  CHECK_NE(ptr, 0u);
  for (size_t i = 0; i < length; ++i) {
    new(reinterpret_cast<void*>(ptr + i * method_size)) ArtMethod;
  }
  return reinterpret_cast<ArtMethod*>(ptr);
}
```

可以看到，ptr 是这个方法数组的指针，而方法是连续创建出来排列在这个方法数组中的。这时只是分配出内存空间，还没对 ArtMethod 的各个成员赋值，不过这并不影响观察 ArtMethod 的空间结构，Artmethod 空间结构如图 2-3 所示。

仅供非商业用途或交流学习使用

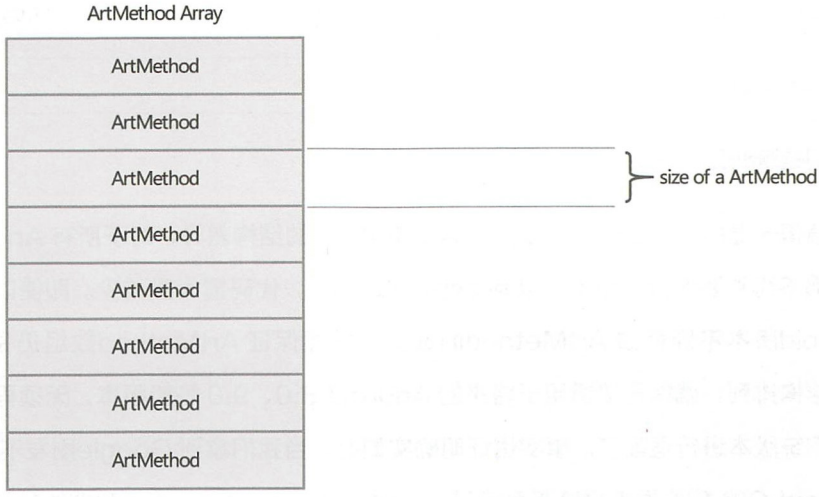


图 2-3 ArtMethod Array 结构

正是这里给了我们启示，ArtMethod 是紧密排列的，所以一个 ArtMethod 的大小，不就是相邻两个 ArtMethod 的起始地址的差值吗?

正是如此。我们就从这个排列特点入手，自己构造一个类，以一种巧妙的方式获取到这个差值。

```
public class NativeStructsModel {
    final public static void f1() {}
    final public static void f2() {}
}
```

由于 f1() 和 f2() 都是 static 方法，所以都属于 direct ArtMethod Array。由于 NativeStructsModel 类中只存在这两个方法，因此它们肯定是相邻的。

那么就可以在 JNI 层取得它们的地址差值：

```
    size_t firMid = (size_t) env->GetStaticMethodID(nativeStructsModelClazz,
                        "f1", "()V");
    size_t secMid = (size_t) env->GetStaticMethodID(nativeStructsModelClazz,
                        "f2", "()V");
    size_t methSize = secMid - firMid;
```

然后，就以这个 `methSize` 作为 `sizeof(ArtMethod)`，代入之前的代码。

```
memcpy(smeth, dmeth, methSize);
```

问题就迎刃而解了。

值得一提的是，由于忽略了底层 ArtMethod 的结构差异，对于所有 Android 版本都不再需要区分，而统一以 `memcpy` 实现即可，代码量大量减少。即使以后的 Android版本不断修改 ArtMethod的成员，只要保证 ArtMethod数组仍是以线性结构排列，就能直接适用于将来的 Android 8.0、9.0 等新版本，无须再针对新的系统版本进行适配了。事实也证明确实如此，当我们拿到 Google刚发不久的 Android O(8.0)开发者预览版系统时，Hotfix demo 直接就能顺利地加载补丁运行起来，并没有做任何适配工作，稳定性极好。

2.2.2 访问权限的问题

(1) 方法调用时的权限检查

看到这里，你可能会产生疑惑：我们只是替换了 ArtMethod 的内容，但替换方法的所属类，和原有方法的所属类，是不同的类型，被替换的方法有权限访问这个类的其他 private 方法吗?

以这段简单的代码为例：

```
public class Demo {
    Demo() {
        func();
    }

    private void func() {
    }
}
```

Demo 构造函数调用私有函数 `func` 所对应的 Dex Code 和 Native Code 为：

```
    void com.patch.demo.Demo.<init>() (dex_method_idx=20628)
     DEX CODE:
       ... ...
       0x0003: 7010 9550 0000              | invoke-direct {v0}, void com.patch.
demo.Demo.func() // method@20629
       ... ...

     CODE: (code_offset=0x006fd86c size_offset=0x006fd868 size=140)...
       ... ...
       0x006fd8c4: f94003e0  ldr x0, [sp]              ; x0 = <init> 的
       ArtMethod*
       0x006fd8c8: b9400400  ldr w0, [x0, #4]          ; w0 = dex_cache_
       resolved_methods_
       0x006fd8cc: d2909710  mov x16, #0x84b8          ; x16 = 0x84b8
       0x006fd8d0: f2a00050  movk x16, #0x2, lsl #16   ; x16 = 0x84b8 + 0x20000
       = 0x284b8 = (20629 + 2) * 8,
                                                       ; 也就是 Demo.func 的
       ArtMethod* 相对于表头 dex_cache_resolved_methods_ 的偏移
       0x006fd8d4: f8706800  ldr x0, [x0, x16]         ; 得到 Demo.func 的
       ArtMethod*
       0x006fd8d8: f940181e  ldr lr, [x0, #48]         ; 取得其 entry_point_from_
       quick_compiled_code_
       0x006fd8dc: d63f03c0  blr lr                    ; 跳转执行
       ... ...
```

这个调用逻辑和之前 Activity 的例子大同小异，需要注意的地方是，在构造函数调用同一个类的私有方法 `func` 时，没有做任何权限检查。也就是说，这时即使把 `func` 方法偷梁换柱，也能直接跳过去正常执行而不会报错。

可以推测，在 dex2oat 生成 AOT 机器码时是有做一些检查和优化的，由于在 dex2oat 编译机器码时确认了两个方法同属一个类，所以机器码中就不存在权限检查的相关代码。

（2）同包名下的权限问题

但是，并非所有方法都可以这么顺利地进行访问。我们发现补丁中的类在访问同包名下的类时，会报出访问权限异常：

```
Caused by: java.lang.IllegalAccessError:
Method 'void com.patch.demo.BaseBug.test()' is inaccessible to class 'com.
```

```
patch.demo.MyClass' (declaration of 'com.patch.demo.MyClass'
Appears in /data/user/0/com.patch.demo/files/baichuan.fix/patch/patch.jar)
```

虽然 com.patch.demo.BaseBug 和 com.patch.demo.MyClass 是同一个包 com.patch.demo 下面的，但是由于我们替换了 com.patch.demo.BaseBug.test，而这个替换了的 BaseBug.test 是从补丁包的 Classloader 中加载的，与原有的 base 包就不是同一个 Classloader 了，这样就导致两个类无法被判别为同包名。具体的校验逻辑是在虚拟机代码的 Class::IsInSamePackage 中：

文件：android-6.0.1_r62/art/runtime/mirror/class.cc。

```
bool Class::IsInSamePackage(Class* that) {
  Class* klass1 = this;
  Class* klass2 = that;
  if (klass1 == klass2) {
    return true;
  }
  // Class loaders must match.
  if (klass1->GetClassLoader() != klass2->GetClassLoader()) {
    return false;
  }
  // Arrays are in the same package when their element classes are.
  while (klass1->IsArrayClass()) {
    klass1 = klass1->GetComponentType();
  }
  while (klass2->IsArrayClass()) {
    klass2 = klass2->GetComponentType();
  }
  // trivial check again for array types
  if (klass1 == klass2) {
    return true;
  }
  // Compare the package part of the descriptor string.
  std::string temp1, temp2;
  return IsInSamePackage(klass1->GetDescriptor(&temp1), klass2-
>GetDescriptor(&temp2));
}
```

关键点在于，Class loaders must match 这行注释。

知道了原因就可以解决问题，我们只要设置新类的 Classloader 为原来类就可以了。而这一步同样不需要在 JNI 层构造底层的结构，只需要通过反射进行设置。这

样仍旧能够保证良好的兼容性。

实现代码如下：

```
Field classLoaderField = Class.class.getDeclaredField("classLoader");
classLoaderField.setAccessible(true);
classLoaderField.set(newClass, oldClass.getClassLoader());
```

这样就解决了同包名下的访问权限问题。

（3）反射调用非静态方法产生的问题

当一个非静态方法被热替换后，在反射调用这个方法时，会抛出异常。

比如下面这个例子：

```
// BaseBug.test 方法已经被热替换了
... ...

BaseBug bb = new BaseBug();
Method testMeth = BaseBug.class.getDeclaredMethod("test");
testMeth.invoke(bb);
```

invoke 的时候就会抛出异常：

```
Caused by: java.lang.IllegalArgumentException:
  Expected receiver of type com.patch.demo.BaseBug,
  but got com.patch.demo.BaseBug
```

这里面，expected receiver 的 BaseBug 和 got 的 BaseBug，虽然都叫 com.patch.demo.BaseBug，但却是不同的类。

前者是被热替换的方法所属的类，由于我们把它的 ArtMethod 的 declaring_class_ 替换了，因此就是新的补丁类。而后者作为被调用的实例对象 bb 的所属类，是原有的 BaseBug。两者是不同的。

在反射 invoke 这个方法时，在底层会调用到 InvokeMethod：

```
jobject InvokeMethod(const ScopedObjectAccessAlreadyRunnable& soa, jobject
javaMethod,
                     jobject javaReceiver, jobject javaArgs, size_t
num_frames) {
      ... ...

      if (!VerifyObjectIsClass(receiver, declaring_class)) {
        return nullptr;
      }

      ... ...
```

这里面会调用 VerifyObjectIsClass 函数做验证。

```
inline bool VerifyObjectIsClass(mirror::Object* o, mirror::Class* c) {
  if (UNLIKELY(o == nullptr)) {
    ThrowNullPointerException("null receiver");
    return false;
  } else if (UNLIKELY(!o->InstanceOf(c))) {
    InvalidReceiverError(o, c);
    return false;
  }
  return true;
}
```

o 表示 Method.invoke 传入的第一个参数，也就是作用的对象。

c 表示 ArtMethod 所属的类型。

因此，只有 o 是 c 的一个实例才能够通过验证，才能继续执行后面的反射调用流程。

由此可知，这种热替换方式所替换的非静态方法，在进行反射调用时，由于 VerifyObjectIsClass 时旧类和新类不匹配，就会导致校验不通过，从而抛出上面那个异常。

那为什么只有方法是非静态才有这个问题呢？因为如果是静态方法，是在类的级别直接进行调用的，就不需要接收对象实例作为参数。所以就没有这方面的检查了。

对于这种反射调用非静态方法的问题，我们会采用另一种冷启动机制应对，后文会说明如何解决此类问题。

2.2.3 即时生效所带来的限制

除了反射的问题，像本方案以及 Andfix 这样直接在运行期修改底层结构的热修复方案，都存在着一个限制，那就是只能支持方法的替换。而对于补丁类里面存在方法增加或减少，以及成员字段的增加或减少的情况，都是不适用的。

原因是这样的，一旦补丁类中出现了方法的增加或减少，就会导致这个类以及整个 Dex 的方法数的变化。方法数的变化伴随着方法索引的变化，这样在访问方法时就无法正常地索引到正确的方法了。

而如果字段发生了增加或减少，和方法变化的情况一样，所有字段的索引都会发生变化。并且更严重的问题是，如果在程序运行中某个类突然增加了一个字段，那么对于原有的这个类的实例，它们还是原来的结构，这是无法改变的。而新方法使用到这些旧的实例对象时，访问新增字段就会产生不可预期的结果。

不过新增一个完整的、原有包里面不存在的新类是可以的，这个不受限制。

总之，只有以下两种情况是不适用的：

- 引起原有类中发生结构变化的修改；
- 修复了的非静态方法会被反射调用。

而对于其他情况，这种方式的热修复都可以任意使用。

虽然有一些使用限制，但一旦满足使用条件，这种热修复方式是十分出众的，它补丁小、加载迅速、能够实时生效无须重新启动 App，并且具有完美的设备兼容性。针对较小程度的修改可以采用本文这种即时生效的热修复方案，并且可以结合资源修复，做到资源和代码的即时生效。而如果触及了上面提到的热替换使用限制，**对于比较大的代码改动以及被修复方法反射调用情况**，Sophix 也提供了另一种完整的 dex 修复机制（将在 3.1 节进行讲解），不过是需要 App 重新冷启动，来发挥其更加完善的修复及更新功能。从而做到无感知的应用更新。

2.3 编译期与语言特性的影响

和业界很多热修复方案不同，Sophix 热修复一直秉承粒度小、注重快捷修复、无侵入适合原生工程等原则。因为坚持这些原则，我们在研发过程中遇到很多编译期的问题，这些问题对最终方案的实施和热部署也带来或多或少的影响，令人印象深刻。

本节列举了在项目实战中遇到的一些挑战，这些都是 Java 语言在编译实现上的一些特点，虽然这些特点与热修复没有直接关系，但深入研究它们对 Android 及 Java 语言的理解都颇有脾益。

2.3.1 内部类编译

有时候我们会发现，在修改外部类某个方法逻辑为访问内部类的某个方法时，最后打出来的补丁包竟然提示新增了一个方法，这真的很匪夷所思。所有有必要了解下内部类在编译期间是怎么工作的，首先我们要知道**内部类在编译期会被编译为跟外部类一样的顶级类。**

(1) 静态内部类 / 非静态内部类的区别

静态内部类 / 非静态内部类的区别大家应该都很熟悉，非静态内部类持有外部类的引用，静态内部类不持有外部类的引用。所以在 Android 性能优化中建议 Handle 的实现尽量使用静态内部类，防止外部 Activity 类不能被回收导致可能的 OOM。我们反编译为 smali 比较两者的不同点：

```
// 静态内部类
# direct methods
.method constructor <init>()V
    return-void
.end method

// 非静态内部类，编译期间会自动合成 this$0 域表示的就是外部类的引用
.field final synthetic this$0:Lcom/taobao/patch/demo/DexFixDemo;
# direct methods
```

```
.method constructor <init>(Lcom/taobao/patch/demo/DexFixDemo;)Via
    .locals 1
    .param p1, "this$0"    # Lcom/taobao/patch/demo/DexFixDemo;

    iput-object p1, p0, Lcom/taobao/patch/demo/DexFixDemo$A;->this$0:Lcom/
taobao/patch/demo/DexFixDemo;
    return-void
.end method
```

（2）内部类和外部类互相访问

既然内部类实际上跟外部类一样都是顶级类，既然都是顶级类，那是不是意味着对方私有的 method/field 是无法被访问得到的，事实上外部类为了访问内部类私有的域 / 方法，编译期间自动会为内部类生成 `access$` 数字编号相关方法：

```
public class BaseBug {
    public void test(Context context) {
        InnerClass inner = new InnerClass("old apk");
        Toast.makeText(context.getApplicationContext(), inner.s, Toast
            .LENGTH_SHORT).show();
    }
    class InnerClass {
        private String s;
        private InnerClass(String s) {
            this.s = s;
        }
    }
}
```

此时外部类 BaseBug 为了能访问到内部类 InnerClass 的私有域 s，编译器会自动为 `InnerClass` 这个内部类合成 `access$100` 方法，这个方法的实现简单返回私有域 s 的值。同样的如果此时匿名内部类需要访问外部类的私有属性 / 方法时，那么外部类也会自动生成 `access$**` 相关方法提供给内部类访问使用。

（3）热部署解决方案

所以有这样一种场景，打补丁前的 test 方法没访问 inner.s，打补丁后的 test 方法访问了 inner.s，那么补丁工具最后检测到了新增的 `access$100` 方法。那么我们

只要防止生成 access$** 相关方法，就能走热部署方案，也就是底层替换方式热修复。所以只要满足以下条件，就能避免编译器自动生成 access$** 相关方法：

- 一个外部类如果有内部类，把所有 method/field 的私有访问权限改成 protected 或 public 或者默认访问权限。
- 同时把内部类的所有 method/field 的私有访问权限改成 protected 或 public 或者默认访问权限。

2.3.2 匿名内部类编译

匿名内部类其实也是个内部类，所以自然也有 2.3.1 节中说明情况的影响，但是我们发现新增一个匿名类（补丁热部署模式下是允许新增类），同时规避 2.3.1 节中的情况，但是匪夷所思的是仍然提示了 method 的新增，所以接下来了解匿名内部类跟非匿名内部类的不同，并且有怎么样的特殊性。

（1）匿名内部类编译命名规则

匿名内部类顾名思义就是没名字的。匿名内部类的名称格式一般是**外部类 $ 数字编号**，后面的数字编号，是编译期根据该匿名内部类在外部类中出现的先后关系，依次累加命名的。

```
public class DexFixDemo {
    public static void test(Context context) {
        /*new DialogInterface.OnClickListener(){
            @Override
            public void onClick(DialogInterface dialog, int which) {
                Log.d("BCHotfix", "OnClickListener");
            }
        };*/

        new Thread("thread-1") {
            @Override
            public void run() {
                Log.d("BCHotfix", "thread-1 thread");
            }
```

```
        }.start();
    }
}
```

修复后的 APK 新增 `DialogInterface.OnClickListener` 这么一个匿名内部类，但是最后补丁工具发现新增了 `onClick` 方法，因为打补丁前只有一个 Thread 匿名内部类，此时该类的名称是 `DexFixDemo$1`，然后打补丁后在 test 方法中新增了 `DialogInterface.OnClickListener` 的匿名内部类。此时 DialogInterface.OnClickListener 匿名内部类名称是 `DexFixDemo$1`，Thread 匿名内部类名称是 `DexFixDemo$2`，所以前后 DexFixDemo$1 类进行对比差异，这个时候已经完全乱套了。同样的道理，**减少一个匿名内部类也存在相同的情况**。

(2) 热部署解决方案

新增或减少匿名内部类，实际上对于热部署来说是无解的，因为补丁工具拿到的是已经编译后的 .class 文件，所以根本没法去区分是 DexFixDemo$1 或是 DexFixDemo$2 类。所以在这种情况下，如果有补丁热部署的需求，应该极力避免插入一个新的匿名内部类。当然如果**匿名内部类是插入到外部类的末尾，那么是允许的**。

2.3.3 有趣的域编译

(1) 静态 field，非静态 field 编译

实际上在热部署方案中除了不支持 method/field 的新增，同时也不支持 `<clinit>` 的修复，这个方法会在 Dalvik 虚拟机中类加载的时候进行类初始化调用。在 Java 源码中本身并没有 clinit 这个方法，这个方法是 Android 编译器自动合成的。通过测试发现，静态 field 的初始化和静态代码块实际上就会被编译器编译在 `<clinit>` 这个方法中，所以我们有必要去了解一下 field/代码块到底是怎么编译的。

来看个简单的示例：

```
public class DexFixDemo {
    {
        i = 2;
    }
    private int i = 1;

    private static int j = 1;
    static {
        j = 2;
    }
}
```

反编译为 smali 看下：

```
.method static constructor <clinit>()V // 类初始化方法
    const/4 v0, 0x1
    sput v0, Lcom/taobao/patch/demo/DexFixDemo;->j:I
    const/4 v0, 0x2
    sput v0, Lcom/taobao/patch/demo/DexFixDemo;->j:I
    return-void
.end method

.method public constructor <init>()V // 构造方法
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V // 首先调用父类的默认构造函数
    const/4 v0, 0x2
    iput v0, p0, Lcom/taobao/patch/demo/DexFixDemo;->i:I
    const/4 v0, 0x1
    iput v0, p0, Lcom/taobao/patch/demo/DexFixDemo;->i:I
    return-void
.end method
```

（2）静态 field 初始化，静态代码块

上面的示例中，能够很明显看到静态 field 初始化和静态代码块被编译器翻译在 `<clinit>` 方法中。**静态代码块和静态域初始化在 `clinit` 中的先后关系就是两者出现在源码中的先后关系**，所以上述示例中最后 j==2。前面说过，类加载进行类初始化的时候，会去调用 clinit 方法，一个类仅加载一次。以下三种情况都会尝试去加载一个类：

1. 创建一个类的对象 (new-instance 指令)；
2. 调用类的静态方法 (invoke-static 指令)；

3. 获取类的静态域的值 (sget 指令)。

首先判断这个类有没有被加载过，如果没有被加载过，执行 `dvmResolveClass->dvmLinkClass->dvmInitClassr` 的流程，类的初始化时在 `dvmInitClass` 中。dvmInitClass 这个函数首先会尝试会对父类进行初始化，然后调用本类的 clinit 方法，所以此时静态 field 得到初始化并且静态代码块得到执行。

(3)非静态 field 初始化，非静态代码块

上面的示例中，能够很明显地看到非静态 field 初始化和非静态代码块被编译器翻译在 `<init>` 默认无参构造函数中。非静态 field 和非静态代码块在 init 方法中的先后顺序也跟两者在源码中出现的顺序一致，所以上述示例中最后 i==1。**实际上如果存在有参构造函数，那么每个有参构造函数都会执行一个非静态域的初始化和非静态代码块。**

> 构造函数会被 Android 编译器自动翻译成 <init> 方法

前面介绍过 clinit 方法在类加载初始化的时候被调用，那么 `<init>` 构造函数方法肯定是对类对象进行初始化时候被调用的，简单来说创建一个对象就会对这个对象进行初始化，并调用这个对象相应的构造函数，看下这行代码 `String s = new String("test")` 编译之后的样子。

```
new-instance v0, Ljava/lang/String;
invoke-direct {v0}, Ljava/lang/String;-><init>()V
```

首先执行 `new-instance` 指令，主要为对象分配堆内存空间，同时如果类之前没加载过，尝试加载类。然后执行 `invoke-direct` 指令调用类的 init 构造函数方法执行对象的初始化。

(4)热部署解决方案

由于不支持 `<clinit>` 方法的热部署，所以任何静态 field 初始化和静态代码块

的变更都会被编译到 clinit 方法中，导致最后热部署失败，只能冷启动生效。如上所见，非静态 field 和非静态代码块的变更被编译到 `<init>` 构造函数中，热部署模式下只是视为一个普通方法的变更，此时对热部署是没有影响的。

2.3.4 final static 域编译

`final static` 域首先是一个静态域，所以我们自然认为其会被编译到 clinit 方法中，所以在自然热部署下也是不能变更。但是测试发现，`final static` 修饰的基本类型或 String 常量类型，匪夷所思的竟然没有被编译到 clinit 方法中，见以下分析。

(1) final static 域编译规则

final static 即静态常量域，看下 final static 域被编译后的样子：

```
public class DexFixDemo
    static Temp t1 = new Temp();
    final static Temp t2 = new Temp();

    final static String s1 = new String("heihei");
    final static String s2 = "haha";

    static int i1 = 1;
    final static int i2 = 2;
}
```

看下反编译得到的 smali 文件：

```
.field static i1:I = 0x0
.field static final i2:I = 0x2
.field static final s1:Ljava/lang/String;
.field static final s2:Ljava/lang/String; = "haha"
.field static t1:Lcom/taobao/patch/demo/Temp;
.field static final t2:Lcom/taobao/patch/demo/Temp;

# direct methods
.method static constructor <clinit>()V
    new-instance v0, Lcom/taobao/patch/demo/Temp;
    invoke-direct {v0}, Lcom/taobao/patch/demo/Temp;-><init>()V
    sput-object v0, Lcom/taobao/patch/demo/DexFixDemo;->t1:Lcom/taobao/patch/
```

```
    demo/Temp;

    new-instance v0, Lcom/taobao/patch/demo/Temp;
    invoke-direct {v0}, Lcom/taobao/patch/demo/Temp;-><init>()V
    sput-object v0, Lcom/taobao/patch/demo/DexFixDemo;->t2:Lcom/taobao/patch/
    demo/Temp;

    new-instance v0, Ljava/lang/String;
    const-string v1, "heihei"
    invoke-direct {v0, v1}, Ljava/lang/String;-><init>(Ljava/lang/String;)V
    sput-object v0, Lcom/taobao/patch/demo/DexFixDemo;->s1:Ljava/lang/String;
    const/4 v0, 0x1
    sput v0, Lcom/taobao/patch/demo/DexFixDemo;->i1:I

    return-void
.end method
```

我们发现，`final static int i2 = 2` 和 `final static String s2 = "haha"` 这两个静态域竟然没被初始化。其他的非 final 静态域均在 clinit 函数中得到初始化。这里注意下 `"haha"` 和 `new String("heihei")` 的区别，前者是字符串常量，后者是引用类型。那这两个 final static 域 (i2 和 s2) 究竟在何处得到初始化？事实上，类加载初始化 dvmInitClass 在执行 clinit 方法之前，首先会先执行 `initSFields`，这个方法的作用主要就是给 static 域赋予默认值。如果是引用类型，那么默认初始值为 NULL。010Editor 工具查看 dex·文件结构，我们能看到在 dex 的类定义区，每个类下面都有这么一段数据，如图 2-4 中的 `encoded_array_item`。

▼ struct class_def_item class_def[1363]	public com.taobao.patch.demo.DexFixDemo
int64 pos	455148
uint class_idx	(0x984) com.taobao.patch.demo.DexFixDemo
enum ACCESS_FLAGS access_flags	(0x1) ACC_PUBLIC
uint superclass_idx	(0x9DE) java.lang.Object
uint interfaces_off	0
uint source_file_idx	(0x868) "DexFixDemo.java"
uint annotations_off	0
uint class_data_off	711815
▶ struct class_data_item class_data	6 static fields, 0 instance fields, 3 direct methods, 0 virtual methods
uint static_values_off	3317723
▼ struct encoded_array_item static_values	4 items: [int: 0, int: 2, NULL: NULL, string: "haha"]
▼ struct encoded_array value	[int: 0, int: 2, NULL: NULL, string: "haha"]
▶ struct uleb128 size	0x4
▶ struct encoded_value values[0]	int: 0
▶ struct encoded_value values[1]	int: 2
▶ struct encoded_value values[2]	NULL: NULL
▶ struct encoded_value values[3]	string: "haha"

图 2-4 encoded_array_item 结构

上述代码示例中，那块区域有4个默认初始值，分别是`t1==NULL`，`t2==NULL`，`s1==NULL`，`s2=="haha"`，`i1==0`，`i2==2`。其中`t1/t2/s2/i1`在initSFields中首先赋值了默认初始化值，然后在随后的clinit中赋值了程序设置的值。而`i2/s2`在initSFields得到的默认值就是程序中设置的值。

现在我们知道了static和final static修饰field的区别了，简单来说：

- final static修饰的**原始类型和`String`类型域(非引用类型)**，在并不会被编译在clinit方法中，而是在类初始化执行initSFields方法时得到了初始化赋值。
- final static修饰的**引用类型**，初始化仍然在clinit方法中。

(2) final static域优化原理

另外一方面，我们经常会看到Android性能优化相关文档中介绍过，如果一个field是常量，那么推荐尽量使用`static final`作为修饰符。**很明显这句话不太对，得到优化的仅仅是`final static`原始类型和`String`类型域(非引用类型)，如果是引用类型，实际上不会得到任何优化的**。还是接着上面的示例，Temp直接引用DexFixDemo的static变量：

```
class Temp {
    public static void test() {
        int i1 = DexFixDemo.i1;
        int i2 = DexFixDemo.i2;

        Temp t1 = DexFixDemo.t1;
        Temp t2 = DexFixDemo.t2;

        String s1 = DexFixDemo.s1;
        String s2 = DexFixDemo.s2;
    }
}
```

看下反编译后的Temp.smali文件：

```
.method public static test()V
    sget v0, Lcom/taobao/patch/demo/DexFixDemo;->i1:I
```

```
    .local v0, "i1":I
    const/4 v1, 0x2
    .local v1, "i2":I
    sget-object v4, Lcom/taobao/patch/demo/DexFixDemo;->t1:Lcom/taobao/patch/
    demo/Temp;
    .local v4, "t1":Lcom/taobao/patch/demo/Temp;
    sget-object v5, Lcom/taobao/patch/demo/DexFixDemo;->t2:Lcom/taobao/patch/
    demo/Temp;
    .local v5, "t2":Lcom/taobao/patch/demo/Temp;
    sget-object v2, Lcom/taobao/patch/demo/DexFixDemo;->s1:Ljava/lang/String;
    .local v2, "s1":Ljava/lang/String;
    const-string v3, "haha"
    .local v3, "s2":Ljava/lang/String;
    return-void
.end method
```

首先来看下 Temp 怎么获取 `DexFixDemo.i2`（final static 域），直接通过 `const/4` 指令：

```
const/4 vA, #+B // 前一个字节是 opcode，后一个字节前 4 位是寄存器 v1，后 4 位就是立即数的
值 "0x02"

HANDLE_OPCODE(OP_CONST_4 /*vA, #+B*/) {
    s4 tmp;
    vdst = INST_A(inst);
    tmp = (s4) (INST_B(inst) << 28) >> 28;  // sign extend 4-bit value
    ILOGV("|const/4 v%d,#0x%02x", vdst, (s4)tmp);
    SET_REGISTER(vdst, tmp);
}
FINISH(1);
OP_END
```

`const/4` 指令的执行过程很简单，操作数在 dex 文件中的位置就是在 opcode 后一个字节。怎么获取 `DexFixDemo.i1`（非 final 域），通过 `sget` 指令。

```
sget vAA, field@BBBB /* 前一个字节是 opcode，后一个字节是寄存器 v0，后两个字节是
DexFixDemo.i1 这个 field 在 dex 文件结构中 field 区的索引值 */

#define HANDLE_SGET_X(_opcode, _opname, _ftype, _regsize)
HANDLE_OPCODE(_opcode /*vAA, field@BBBB*/) {
    StaticField* sfield;
    vdst = INST_AA(inst);
    ref = FETCH(1);
    sfield = (StaticField*)dvmDexGetResolvedField(methodClassDex, ref);
    if (sfield == NULL) {
```

```
            EXPORT_PC();
            sfield = dvmResolveStaticField(curMethod->clazz, ref);
            if (sfield == NULL)
                GOTO_exceptionThrown();
            if (dvmDexGetResolvedField(methodClassDex, ref) == NULL) {
                JIT_STUB_HACK(dvmJitEndTraceSelect(self,pc));
            }
        }
        SET_REGISTER##_regsize(vdst, dvmGetStaticField##_ftype(sfield));
    }
    FINISH(2);
```

首先调用 `dvmDexGetResolvedField` 并判断这个域之前是否被解析过，如果没有被解析过，调用 `dvmResolveStaticField` 尝试解析域，如果这个静态域所在的类没有被解析还会调用 `dvmResolveClass` 解析类。然后拿到这个 sfield 静态域，调用 `dvmGetStaticFieldInt(sfield)`，得到这个静态域的值。可见此时 `sget` 指令比 `const/4` 指令的解析过程要复杂，所以 final static 基本类型可以得到优化。

final static String 类型引用 `const-string` 指令的解析执行速度比 `sget-object` 指令的解析要快一些。本质上跟 final static 基本类型差不多，只是有略微的区别，final static String 类型的变量，编译期间引用会被优化成 const-string 指令，因为 `const/4` 获取的值是个**立即数**，但是 `const-string` 指令获取的只是**字符串常量在 dex 文件结构中字符常量区的索引** ID，所以需要额外的一次字符串查找。dex 文件中有一块区域存储着程序所有的字符串常量，最终这块区域会被虚拟机完整加载到内存中，这块区域也就是通常说的“字符串常量区”内存。

final static 引用类型没有得到优化，是因为不管是不是 final，最后都是通过 `sget-object` 指令去获取该值，所以此时实际上从虚拟机运行性能方面来说得不到任何优化，此时 final 的作用，仅仅是让编译器能在编译期间检测到该 final 域有没有被修改。final 域修改过在编译期间就会直接报错。

(3)热部署解决方案

最后所有可以得到的结论：

- 修改 final static 基本类型或者 String 类型域（非引用类型域），由于在编译期间引用到基本类型的地方被立即数替换，引用到 String 类型（非引用类型）的地方被常量池索引 ID 替换，所以在热部署模式下，最终所有引用到该 final static 域的方法都会被替换。实际上此时仍然可以执行热部署方案。
- 修改 final static 引用类型域，是不允许的，因为这个 field 的初始化会被编译到 clinit 方法中，所以此时没法走热部署。

2.3.5 有趣的方法编译

(1)应用混淆方法编译

除了以上的内部类和匿名内部类可能会造成 method 新增之后，我们发现项目如果应用了混淆方法编译，可能导致方法的内联和裁剪，那么最后也可能导致 method 的新增或减少，以下介绍在哪些场景中会造成方法的内联或裁剪。

(2)方法内联

实际上有好几种情况可能导致方法被内联掉。

- 方法没有被其他任何地方引用，毫无疑问，该方法会被内联掉。
- 方法足够简单，比如一个方法的实现就只有一行代码，该方法会被内联掉，那么任何调用该方法的地方都会被该方法的实现替换掉。
- 方法只被一个地方引用，这个地方会被方法的实现替换掉。

举个简单的例子进行说明下。

```
public class BaseBug {
    public static void test(Context context) {
```

```
        Toast.makeText(context.getApplicationContext(), "old apk...", Toast.
LENGTH_SHORT).show();
        print("haha");
    }
    public static void print(String s) {
        ... ...
        Log.d("BaseBug", s);
    }
}
```

此时假如 print 方法足够复杂，同时只在 test 方法中被调用，假设 test 方法没有被内联，print 方法由于只有一个地方调用会被内联。查看下生成的 mApping.txt 文件，印证了这个结论，此时没有看到 print 方法的映射，说明这个方法被内联掉了。

```
com.taobao.hotfix.demo.BaseBug -> com.taobao.hotfix.demo.a:
    void test(android.content.Context) -> a
```

如果恰好将要打补丁的一个方法调用了 print 方法，那么 print 方法被调用了两次，在新的 APK 中不会被内联，补丁工具检测到新增了 print 方法。那么该补丁只能走冷启动方案。

(3)方法裁剪

```
public class BaseBug {
    public static void test(Context context) {
        Log.d("BaseBug", "test");
    }
}
```

查看下生成的 mApping.txt 文件：

```
com.taobao.hotfix.demo.BaseBug -> com.taobao.hotfix.demo.a:
    void test$faab20d() -> a
```

此时 test 方法 context 参数没被使用，所以 test 方法的 context 参数被裁剪，混淆任务首先生成 `test$faab20d()` 裁剪过后的无参方法，然后再混淆。所以如果将要打补丁该 test 方法，同时恰好用到了 context 参数，那么 test 方法的 context 参数不会被裁剪，那么补丁工具检测到新增了 `test(context)` 方法。那么该补丁只

能走冷启动方案。

怎么让该参数不被裁剪？当然是有办法的，不让编译器在优化的时候认为引用了一个无用的参数就好了，可以采取的方法很多，这里介绍一种最有效的方法：

```
public static void test(Context context) {
    if (Boolean.FALSE.booleanValue()) {
        context.getApplicationContext();
    }
    Log.d("BaseBug", "test");
}
```

注意：这里不能用基本类型 false，必须用包装类 Boolean，因为如果使用基本类型 if 语句也很可能会被优化掉的。

（4）热部署解决方案

实际只要混淆配置文件加上 `-dontoptimize` 项就不会去做方法的裁剪和内联。在一般情况下项目的混淆配置都会使用到 Android SDK 默认的混淆配置文件 `proguard-android-optimize.txt` 或者 `proguard-android.txt`，两者的区别就是后者应用了 -dontoptimize 这一项配置而前者没有应用。

实际上述几个步骤都是可以选的，对混淆库可能给热部署带来的影响进行简单说明，影响主要在 optimization 阶段。

`optimization step:` 进一步优化代码，非入口点的类和方法可以被设置为 private、static 或 final，无用的参数可能被移除，并且一些方法可能会被内联。

可以看到在 optimization 阶段，除了方法的裁剪和内联可能导致方法的新增或减少之外；还可能把方法的修饰符优化成 private/static/final 属性，那么此时补丁工具发现这个方法的实现发生了变更，会对这个方法做毫无意义的打包。所以在补丁热部署模式下，混淆配置最好都加上 `-dontoptimize` 这项。

preverification step: 针对 .class 文件的预校验，在 .class 文件中加上 StackMa/StackMapTable 信息，这样 Hotspot VM 在类加载时执行类校验阶段会省去一些步骤，因此类加载将会更快。

我们知道 Android 虚拟机执行的是 dex 文件格式，编译期间 dx 工具会把所有的 .class 文件优化成 .dex 文件，所以混淆库的预编译在 Android 中是没有任何意义的，反而会降低打包速度，Android 虚拟机中有自己的一套代码校验逻辑（dvmVerifyClass）。所以 Android 中混淆配置一般都需要加上 `-dontpreverify` 这一项。

还有混淆库对反射的处理也特别有意思，有需要了解的可以查看 proguard.jar 源码。

```
Class clz = Class.forName("com.taobao.test.Temp");
Method method = clz.getDeclaredMethod("test", new Class[]{Context.class,
String.class});
```

com.taobao.test.Temp 类不会被移除，test 方法有可能在 shrinking 阶段被移除或者在 obfuscation 阶段被重命名，最后导致反射调用失败。

```
Method method = com.taobao.test.Temp.class.getDeclaredMethod("test", new Class[]
{Context.class, String.class});
```

com.taobao.test.Temp 类不会被移除，test 方法不会被移除，obfuscation 阶段这个方法和 .class 这一行代码的字节码同时发生变更，所以不会失败。

2.3.6 switch case 语句编译

由于在实现资源修复方案热部署的过程中（第 4 章将进行详解），要做新旧资源 ID 的替换，我们竟然发现存在 switch case 语句中的 ID 不会被替换掉的情况，所以有必要来探索下 `switch case` 语句编译的特殊性。

(1) switch case 语句编译规则

```
public void testContinue() {
    int temp = 2;
    int result = 0;
    switch (temp) {
        case 1:
            result = 1;
            break;
        case 3:
            result = 3;
            break;
        case 5:
            result = 5;
            break;
    }
}
public void testNotContinue() {
    int temp = 2;
    int result = 0;
    switch (temp) {
        case 1:
            result = 1;
            break;
        case 3:
            result = 3;
            break;
        case 10:
            result = 10;
    }
}
```

看下 testContinue/testNotContinue 方法编译出来有何不同。

```
# virtual methods
.method public testContinue()V
    const/4 v1, 0x2
    .local v1, "temp":I
    const/4 v0, 0x0
    .local v0, "result":I
    packed-switch v1, :pswitch_data_0

    :pswitch_0
    return-void
    :pswitch_1
    const/4 v0, 0x1
    :pswitch_2
```

```
    const/4 v0, 0x3
    :pswitch_3
    const/4 v0, 0x5

    :pswitch_data_0
    .packed-switch 0x1
        :pswitch_1
        :pswitch_0
        :pswitch_2
        :pswitch_0
        :pswitch_3
    .end packed-switch
.end method

.method public testNotContinue()V
    const/4 v1, 0x2
    .local v1, "temp":I
    const/4 v0, 0x0

    .local v0, "result":I
    sparse-switch v1, :sswitch_data_0

    :sswitch_0
    const/4 v0, 0x1
    :sswitch_1
    const/4 v0, 0x3
    :sswitch_2
    const/16 v0, 0xa

    :sswitch_data_0
    .sparse-switch
        0x1 -> :sswitch_0
        0x3 -> :sswitch_1
        0xa -> :sswitch_2
    .end sparse-switch
.end method
```

testContinue 方法的 switch case 语句被编译成 packed-switch 指令，testNotContinue 方法的 switch case 语句被编译成 sparse-switch 指令。比较两者间的差异，testContinue 的 switch 语句的 case 项是**连续的几个比较相近的值1、3、5**。所以被编译为 packed-switch 指令，可以看到几个连续的数中间的差值用 :pswitch_0 补齐，:pswitch_0 标签处直接 return-void。testNotContinue 的 switch 语句的 case 项分别是 1、3、10，很明显**不够连续**，所以被编译为 sparse-switch 指令。编译器会决定怎样的值才算是连续的 case。

(2)热部署解决方案

一个资源 ID 肯定是 `const final static` 变量，此时恰好 switch case 语句被编译成 packed-switch 指令，所以这个时候如果不做任何处理就会存在资源 ID 替换不完全的情况。解决这种情况方案其实很简单，修改 smali 反编译流程，碰到 `packed-switch` 指令强转为 `sparse-switch` 指令，`:pswitch_N` 等相关标签指令也需要强转为 `:sswitch_N` 指令。然后做资源 ID 的暴力替换，然后再回编译 smali 为 dex。再做类方法变更的检测，所以就需要经过**反编译 → 资源 ID 替换 → 回编译**的过程，这也会使打补丁变得稍慢一些。

2.3.7 泛型编译

泛型是 Java5 才开始引入的，我们发现泛型的使用，也可能导致 method 的新增，所以是时候深入了解一下泛型的编译过程了。

(1)为什么需要泛型

- Java 语言中的**泛型基本上完全在编译器中实现**，由编译器执行类型检查和类型推断，然后生成普通的非泛型的字节码，就是虚拟机完全无感知泛型的存在。这种实现技术称为擦除(erasure)。编译器使用泛型类型信息保证类型安全，然后在生成字节码之前将其清除。
- Java5 才引入泛型，所以扩展虚拟机指令集来支持泛型被认为是无法接受的，因为这会为 Java 厂商升级其 JVM 造成难以逾越的障碍。因此采用了可以完全在编译器中实现的擦除方法。

我们知道了以上两点，其中最重要的就是**类型擦除**的理解，先来通过一个简单的例子说明 Java 为什么需要泛型。Java5 之前，要实现类似“泛型”的功能，由于 Java 类都是以 Object 为最上层的父类别，所以可以用 `Object` 来实现类似“泛型”的功能。

```
public class ObjectFoo {
    private Object foo;
    public void setFoo(Object foo) {
        this.foo = foo;
    }
    public Object getFoo() {
        return foo;
    }
}
```

代码调用示例：

```
    ObjectFoo foo1 = new ObjectFoo();
    foo1.setFoo(new Boolean(true));
    Boolean b = (Boolean) foo1.getFoo();// 正确
    String s = (String) foo1.getFoo();// 运行时，类型转换失败 ClassCastException 异常
```

由于语法上并没有错误，所以编译器检查不出上面的程序有错误，真正的错误要在执行期才会发生，这时恼人的 `ClassCastException` 就会出来搞怪，Boolean 类型当然不能转换为 String 类型。

所以可以看到使用 Object 来实现“泛型”存在一些问题，因为必须要强制类型转换，但很多人可能忘记强制类型转换，或者是强转用错类型（本该用 Boolean 却用了 Integer），然而由于语法上是可以的，所以编译器检查不出错误，因而执行时期就会发生 `ClassCastException`，在 Java5 之后，提出了针对泛型设计的解决方案。**该方案在编译时进行类型安全检测，允许程序员在编译时就能检测到非法的类型**，泛型解决方案如下：

```
public class GenericFoo<T> {
    private T foo;
    public void setFoo(T foo) {
        this.foo = foo;
    }
    public T getFoo() {
        return foo;
    }
}
```

代码调用示例：

```
GenericFoo<Boolean> genericFoo = new GenericFoo();
genericFoo.setFoo(new Boolean(true));
Boolean b = genericFoo.getFoo();// 正确
String s = (String) genericFoo.getFoo();// 编译不通过，incovertiable types
```

很明显此时使用泛型的优势就体现出来了，编译期间就检查到了可能的异常，我们发现 `Boolean b = genericFoo.getFoo()` 这里并不需要做强制类型转换，实际上编译器会在字节码中自动加上强制类型转换。

（2）泛型类型擦除

前面说过，Java 中的泛型基本上都是在编译器这个层次来实现的。在生成的 Java 字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，编译器在编译的时候会去掉，这个过程就称为类型擦除。反编译看下字节码就很容易看到了。

```
.method public getFoo()Ljava/lang/Object;
.method public setFoo(Ljava/lang/Object;)V
```

所以如果此时再定义一个 `setFoo(Object foo)` 方法肯定是行不通的，编译期间报重复方法定义。如果这样的 `<T extends Numble>`，限定了泛型，那么是这样的。

```
.method public getFoo()Ljava/lang/Number;
.method public setFoo(Ljava/lang/Number;)V
```

所以我们知道 `new T()` 这样使用泛型是编译不过的，因为类型擦除会导致实际上是 `new Object()`，所以是错误的。

（3）类型擦除与多态的冲突和解决

```
class A<T> {
    private T t;
```

```
    public T get() {
        return t;
    }
    public void set(T t) {
        this.t = t;
    }
}

class B extends A<Number> {
    private Number n;
    @Override // 跟父类返回值不一样，为什么重写父类 get 方法
    public Number get() {
        return n;
    }

    @Override // 跟父类方法参数类型不一样，为什么重写父类 set 方法
    public void set(Number n) {
        this.n = n;
    }
}

class C extends A {
    private Number n;
    @Override // 跟父类返回值不一样，为什么重写父类 get 方法
    public Number get() {
        return n;
    }
    //@Override 重载父类 get 方法，因为方法参数类型不一样，这里没问题
    public void set(Number n) {
        this.n = n;
    }
}
```

按照前面类型擦除的分析，为什么类 B 的 `set` 和 `get` 方法可以用 `@Override` 而不报错。`@Override` 表明这个方法是重写，我们知道**重写的意思是子类中的方法与父类中的某一方法具有相同的方法名，返回类型和参数表**。但是根据前面的分析，基类 A 由于类型擦除的影响，`set(T t)` 在字节码中实际上是 `set(Object t)`，那么类 B 的方法 `set(Number n)` 方法参数不一样，此时类 B 的 set 方法理论上来说应该重载而不是重写基类的 set 方法。但是我们的本意是进行重写，实现多态，可是类型擦除后，只能变为重载。这样，类型擦除就和多态有了冲突。实际上 JVM 采用了一个特殊的方法，来完成这项重写功能，那就是桥接。看下类 B 的字节码表示：

```
.method public get()Ljava/lang/Number;
.method public bridge synthetic get()Ljava/lang/Object;
    invoke-virtual {p0}, Lcom/taobao/test/B;->get()Ljava/lang/Number;
    move-result-object v0
    return-object v0
.end method

.method public set(Ljava/lang/Number;)V
.method public bridge synthetic set(Ljava/lang/Object;)V
    check-cast p1, Ljava/lang/Number;
    invoke-virtual {p0, p1}, Lcom/taobao/test/B;->set(Ljava/lang/Number;)V
    return-void
.end method
```

我们可以发现编译器自动合成 `set(Ljava/lang/Object;)` 和 `get()Ljava/lang/Object;` 这两个桥接方法来重写父类方法，同时这个两个桥接方法实际上调用 `B.print(Ljava/lang/Number;)` 和 `B.get()Ljava/lang/Number` 这两个重载方法。那么可以得到最终的结论：

- 子类中真正重写基类方法的是编译器自动合成的桥接方法。而类 B 定义的 get 和 set 方法上面的 @Override 只不过是假象，桥接方法的内部实现是去调用自己重写的 print 方法而已。所以，**虚拟机巧妙使用了桥接方法，来解决了类型擦除和多态的冲突。**

这里或许也会有疑问，类 B 中的字节码中的方法 `get()Ljava/lang/Number;` 和 `get()Ljava/lang/Object;` 是同时存在的，这就颠覆了我们的认知，如果是我们自己编写 Java 源代码，这样的代码是无法通过编译器的检查的，方法的重载只能以方法参数而无法以返回类型作为函数重载的区分标准，但是虚拟机却是允许这样操做的，**因为虚拟机通过参数类型和返回类型共同来确定一个方法，所以编译器为了实现泛型的多态允许自己做这个看起来“不合法”的事情**，然后交给虚拟器自己去区别处理。

(4) 泛型类型转换

同时前面我们还留了一个坑，泛型是可以不需要强制类型转换。

```
    //class GenericFoo<T>

    GenericFoo<Boolean> genericFoo1 = new GenericFoo<Boolean>(); /*new
GenericFoo<Boolean>() 其中 <Boolean> 非必须，但是 GenericFoo<Boolean> genericFoo1
的 <Boolean> 是必须的 */
    genericFoo1.setFoo(new Boolean(true));
    Boolean b = genericFoo1.getFoo();// 正确

    GenericFoo genericFoo2 = new GenericFoo();
    genericFoo2.setFoo(new Boolean(true));
    Boolean b = (Boolean) genericFoo2.getFoo();// 必须强制类型转换，否则编译不过
```

代码示例中，第一个不需要强制类型转换，但是第二个必须强制类型转换否则编译期报 incovertiable types 错误。反编译 smali：

```
.method public test()V
    .....
    invoke-virtual {v2}, Lcom/taobao/test/GenericFoo;->getFoo()Ljava/lang/
Object;
    move-result-object v0
    check-cast v0, Ljava/lang/Boolean;

    .....
    invoke-virtual {v3}, Lcom/taobao/test/GenericFoo;->getFoo()Ljava/lang/
Object;
    move-result-object v1
    check-cast v1, Ljava/lang/Boolean;
.end method
```

字节码文件很意外，两者其实并没有什么区别，实际上在编译期间，**编译器发现如果有一个变量的申明加上了泛型的话，编译器自动加上 `check-cast` 类型转换**，而不再需要程序员在源码文件中进行强制类型转换，这里不需要并不意味着不会进行类型转换，可以发现其实类型转换是编译器自动完成了。

（5）热部署解决方案

前面类型擦除中说过，如果由 `B extends A` 变成了 `B extends A<Number>`，那么就可能会新增对应的桥接方法。此时新增了方法，只能走冷部署。在这种情况下，如果要走热部署，应该避免类似上面的那种修复。

另外一方面，实际上泛型方法内部会生成一个 `dalvik/annotation/Signature` 这个系统注解：

```
class A {
    public <A extends Number> void getA(A t) {
        System.out.println("t:" + t);
    }
}

//getA 方法内部注解
.annotation system Ldalvik/annotation/Signature;
    value = {
        "<A:",
        "Ljava/lang/Number;",
        ">(TA;)V"
    }
.end annotation
```

如果现在把方法的签名换成 `<B extends Number> void getA(B t)`，前面说过，泛型类型擦除方法的逻辑实际上没有发生任何变化，但是这个方法的注解却发生了变化。

```
.annotation system Ldalvik/annotation/Signature;
    value = {
        "<B:",
        "Ljava/lang/Number;",
        ">(TB;)V"
    }
.end annotation
```

补丁工具发现这个方法发生了变化（**不会排除注解的变化**），然后对这个方法进行打包，此时打包是多余的没有任何意义的，所以热部署下，需要避免这种会导致浪费性能的修复。

2.3.8 Lambda 表达式编译

Lambda 表达式是 Java7 才引入的一种表达式，类似于匿名内部类，实际上又与匿名内部类有很大的区别，我们发现 Lambda 表达式的使用也可能导致方法的新

增或减少，导致最后走不了热部署模式。所以是时候深入了解一下 Lambda 表达式的编译过程了。

(1) Lambda 表达式编译规则

首先简单介绍下 Lambda 表达式，Lambda 为 Java 添加了缺失的函数式编程特点，Java 现在提供的最接近闭包的概念便是 Lambda 表达式。gradle 就是基于 groovy 存在的大量闭包。函数式接口具有两个主要特征：**它是一个接口，这个接口具有唯一的抽象方法；我们将同时满足这两个特性的接口称为函数式接口**。比如 Java 标准库中的 `java.lang.Runnable` 和 `java.util.Comparator` 都是典型的函数式接口。函数式接口跟匿名内部类的区别如下：

- 关键字 this，匿名类的 this 关键字指向匿名类，而 Lambda 表达式的 this 关键字指向包围 Lambda 表达式的类。
- 编译方式，Java 编译器将 Lambda 表达式编译成类的私有方法，使用 Java7 的 invokedynamic 字节码指令来动态绑定这个方法。Java 编译器将匿名内部类编译成外部类 `$` 数字编号的新类。

通过一个代码示例来讲解 Lambda 表达式，匿名内部类前面已经详细介绍过。

```
public interface TInterface { //自定义函数式接口
    int test(String s);
}
public class Test {
    private static int temp = 2;
    public static void main(String args[]) throws Exception {
        new Thread(() -> {
            System.out.println("java8 Thread lamada...");
        }).start();

        test(s -> temp + 1);
    }
    public static void test(TInterface tInterface) {
        System.out.println("java8 TInterface lamada..." + tInterface.
test(1));
    }
}
```

查看 `javap -p -v Test.class` 反编译 .class 文件，截取关键内容。

```
public class com.demo.Test
  SourceFile: "Test.java"
  InnerClasses:
       public static final #103= #102 of #106; /*Lookup=class java/lang/
invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles */
  BootstrapMethods:
    0: #50 invokestatic java/lang/invoke/LambdaMetafactory.
metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/
lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/
MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
      Method arguments:
        #51 ()V
        #52 invokestatic com/demo/Test.lambda$main$0:()V
        #51 ()V
    1: #50 invokestatic java/lang/invoke/LambdaMetafactory.
metafactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/
lang/invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/
MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
      Method arguments:
        #56 (I)I
        #57 invokestatic com/demo/Test.lambda$main$1:(I)I
        #56 (I)I
Constant pool:
    #3 = InvokeDynamic      #0:#53         //  #0:run:()Ljava/lang/Runnable;
    #6 = InvokeDynamic      #1:#58         //  #1:test:()Lcom/demo/TInterface;
    ....

public static void main(java.lang.String[]) throws java.lang.Exception;
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=1, args_size=1
         0: new           #2                  // class java/lang/Thread
         3: dup
         4: invokedynamic #3, 0              // InvokeDynamic #0:run:()Ljava/
lang/Runnable;
         9: invokespecial #4                  // Method java/lang/
Thread."<init>":(Ljava/lang/Runnable;)V
        12: invokevirtual #5                  // Method java/lang/Thread.
start:()V
        15: invokedynamic #6, 0              // InvokeDynamic #1:test:()Lcom/
demo/TInterface;
        20: invokestatic  #7                  // Method test:(Lcom/demo/
TInterface;)V

private static int lambda$main$1(int);
private static void lambda$main$0();
```

这里我们可以发现以下几点：

- 编译期间自动生成私有静态的 `lambda$main$**(*)` 方法，这个方法的实现其实就是 Lambda 表达式里面的逻辑。
- invokedynamic 指令执行 Lambda 表达式。
- 比较与匿名内部类的区别，发现并没有在磁盘上生成外部类 `$` 数字编号的新类。

那么我们应该思考的是 `invokedynamic` 指令的执行，最后为什么就调用到了 Test 的私有静态 `lambda$main$**(*)` 方法。我们首先看下 JVM 中关于 invokedynamic 指令的介绍：在 Java7 JVM 中增加了一个新的指令 invokedynamic，用于支持动态语言，即允许方法调用可以在运行时指定类和方法，不必在编译的时候确定。字节码中每条 invokedynamic 指令出现的位置称为一个动态调用点，invokedynamic 指令后面会跟一个指向常量池的调用点限定符 (#3，#6)，这个限定符会被解析为一个动态调用点。

从上面的反编译后的内容中可以发现，`invokedynamic` 指令执行时实际上会去调用 `java/lang/invoke/LambdaMetafactory` 的 `metafactory` 静态方法。**这个静态方法实际上会在运行时生成一个实现函数式接口的具体类**，然后具体类会调用 Test 的私有静态 `lambda$main$**(*)` 方法。我们可以通过添加 `-Djdk.internal.lambda.dumpProxyClasses` 这个虚拟机运行参数，那么运行时会将生成的新类的 .class 内容输出到一个文件中，文件内容如下：

```
// $FF: synthetic class
final class Test$$Lambda$1 implements Runnable {
    private Test$$Lambda$1() {
    }
    @Hidden
    public void run() {
        Test.lambda$main$0();
    }
}
// $FF: synthetic class
final class Test$$Lambda$2 implements TInterface {
    private Test$$Lambda$2() {
    }
```

```
    @Hidden
    public int test(int var1) {
        return Test.lambda$main$1(var1);
    }
}
```

Sun/Oracle Hotspot VM 大概就是这么解释 .class 文件中 Lambda 表达式的。那来看下 Android 虚拟机下是怎么解释 Lambda 表达式，探索下两者的异同点。

我们知道 Android 虚拟机**首先通过 `javac` 把源代码编译成 `.class`，然后再通过 `dx` 工具优化成适合移动设备的 `dex` 字节码文件**。但是 Android 中如果要使用新的 Java8 语言特性，还需使用新的 Jack 工具链来替换掉旧的工具链来编译。新的 Android 工具链将 Java 源语言编译成 Android 可读取的 Dalvik 可执行文件字节码，且有自己的 `.jack` 库格式，Jack 是 `Java Android Compiler Kit` 的缩写，它可以将 Java 代码直接编译为 Dalvik 字节码，并负责 `Minification`、`Obfuscation`、`Repackaging`、`Multidexing`、`Incremental compilation`。它试图取代 javac/dx/proguard/jarjar/multidex 库等工具。

以下是构建 Android Dalvik 可执行文件可用的两种工具键的对比。

- 旧版 javac 工具键：

  javac (.java → .class) → dx (.class → .dex)

- 新版 Jack 工具链：

  Jack (.java → .jack → .dex)

上面我们知道为了使用 Java8 中的 Lambda 表达式特性，就必须使用新的 Jack 工具链，新的 Jack 工具链编译产物中是没有 .class 文件。所有我们直接对新的 Jack 工具链编译出来的 .dex 进行反编译，接着使用上面的代码示例。

```
//Test.smali 内容
.method static synthetic lambda$-com_taobao_test_Test_lambda$1()V // 合成的方法
.method static synthetic lambda$-com_taobao_test_Test_lambda$2(I)I // 合成的方法

.method public static main([Ljava/lang/String;)V
    move-object v0, p0
```

```
    .local v0, "args":[Ljava/lang/String;
    new-instance v1, Ljava/lang/Thread;
    new-instance v2, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String__args_LambdaImpl0;
    invoke-direct {v2}, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String__args_LambdaImpl0;-><init>()V
    invoke-direct {v1, v2}, Ljava/lang/Thread;-><init>(Ljava/lang/Runnable;)V
    invoke-virtual {v1}, Ljava/lang/Thread;->start()V

    new-instance v3, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String__args_LambdaImpl1;
    invoke-direct {v3}, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String__args_LambdaImpl1;-><init>()V
    invoke-static {v3}, Lcom/taobao/test/Test;->test(Lcom/taobao/test/
TInterface;)V

    return-void
.end method

//Test$$Lambda$-void_main_java_lang_String__args_LambdaImpl0.smali 内容
.class final synthetic Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String__args_LambdaImpl0;
.super Ljava/lang/Object;
# interfaces
.implements Ljava/lang/Runnable;

# virtual methods
.method public run()V
    invoke-static {}, Lcom/taobao/test/Test;->lambda$-com_taobao_test_Test_
lambda$1()V
    return-void
.end method

//Test$$Lambda$-void_main_java_lang_String__args_LambdaImpl1.smali
```

很明显可以看到 .dex 字节码文件和 .class 字节码文件对 Lambda 表达式处理的异同点。

- 共同点：编译期间都会为外部类合成一个 static 辅助方法，该方法内部逻辑实现 Lambda 表达式。
- 不同点：
  1 .class 字节码中通过 invoke-dynamic 指令执行 Lambda 表达式。而 .dex 字节码中执行 Lambda 表达式跟普通方法调用没有任何区别。

2 .class 字节码中运行时生成新类，.dex 字节码中编译期间生成新类。

（2）热部署解决方案

有了以上知识点做基础，同时知道打补丁是通过反编译为 smali 然后新 APK 跟基线 APK 进行差异对比，得到最后的补丁包。

新增一 Lambda 表达式，会导致外部类新增一个辅助方法，所以此时不支持走热部署方案。

那么如果只是修改 Lambda 表达式内部的逻辑，此时看起来仅仅相当于修改了一个方法，所以此时看起来是允许走热部署的。事实上并非如此。我们忽略了一种情况，Lambda 表达式访问外部类非静态 field/method 的场景。

前面我们知道 .dex 字节码中 Lambda 表达式在编译期间会自动生成新的辅助类。注意该辅助类是非静态的，所以**该辅助类如果为了访问“外部类”的非静态 `field/method` 就必须持有“外部类”的引用。如果该辅助类没有访问“外部类”的非静态 `field/method`，那么就不会持有“外部类”的引用。这里注意这个辅助类和内部类的区别。我们前面说过如果是非 `static` 内部类的话一定会持有外部类的引用的**！下面的示例很容易说明这个问题。

Lambda 表达式没有访问“外部类”的非静态 field/method，可以看到并没有持有 " 外部类 " 引用。

```
public synthetic constructor <init>()V
```

Lambda 表达式访问了“外部类”的非静态 field/method，可以看到持有了“外部类”引用。

```
private synthetic val$this:Lcom/taobao/test/Test; /* 合成的变量 val$this, " 外部
类 " 引用 */
public synthetic constructor <init>(Lcom/taobao/test/Test;)V
    iput-object p1, p0, Lcom/taobao/test/Test$$Lambda$-void_main_java_lang_
String__args_LambdaImpl1;->val$this:Lcom/taobao/test/Test;
```

所以此时存在这样的情况，基线 APK 中 Lambda 表达式中没有访问非静态 field/method，修复后的 APK 中 Lambda 表达式中访问非静态 field/method。那么会导致发现新增 field。此时热部署也会失败。

最后对这一小节进行简短的总结：

- 增加或减少一个 Lambda 表达式会导致类方法比较错乱，所以会导致热部署失败。
- 修改一个 Lambda 表达式，可能导致新增 field，所以此时也会导致热部署失败。

2.3.9 访问权限检查对热替换的影响

2.2.2 节中有提到权限问题对于底层热替换的影响，下面我们就来深入剖析虚拟机下权限控制可能给热修复方案带来的影响，下面代码示例仅演示 Dalvik 虚拟机。

(1) 类加载阶段父类 / 实现接口访问权限检查

一个类的加载过程，必须经历 resolve、link、init 三个阶段，父类或实现接口权限控制检查主要发生在 link 阶段。代码如下：

```
bool dvmLinkClass(ClassObject* clazz){
    ... ...
    if (clazz->status == CLASS_IDX) {
        ... ...
        if (clazz->interfaceCount > 0) {
            for (i = 0; i < clazz->interfaceCount; i++) {
                assert(interfaceIdxArray[i] != kDexNoIndex);
                clazz->interfaces[i] =
                    dvmResolveClass(clazz, interfaceIdxArray[i], false);
                ... ...
                /* are we allowed to implement this interface? */
                if (!dvmCheckClassAccess(clazz, clazz->interfaces[i])) {
                    //检查所有实现的接口的访问权限
                    dvmLinearReadOnly(clazz->classLoader, clazz->interfaces);
                    ALOGW("Interface '%s' is not accessible to '%s'",
                         clazz->interfaces[i]->descriptor, clazz-
                         >descriptor);
```

```
                    dvmThrowIllegalAccessError("interface not accessible");
                    goto bail;
                }
            }
        }
    }
    ... ...
    if (strcmp(clazz->descriptor, "Ljava/lang/Object;") == 0) {
        ... ...
    } else {
        if (clazz->super == NULL) {
            dvmThrowLinkageError("no superclass defined");
            goto bail;
        }
        /* verify */
        else if (!dvmCheckClassAccess(clazz, clazz->super)) {
        //检查父类的访问权限
            ALOGW("Superclass of '%s' (%s) is not accessible",
                clazz->descriptor, clazz->super->descriptor);
            dvmThrowIllegalAccessError("superclass not accessible");
            goto bail;
        }
}
```

上述代码示例可以看到，会依次对当前类实现的接口和父类进行访问权限检查。接下来看 dvmCheckClassAccess 的逻辑：

```
bool dvmCheckClassAccess(const ClassObject* accessFrom,
    const ClassObject* clazz){
    if (dvmIsPublicClass(clazz)) //如果父类是public类，直接return true.
        return true;
    return dvmInSamePackage(accessFrom, clazz);
}
bool dvmInSamePackage(const ClassObject* class1, const ClassObject* class2){
    /* quick test for intra-class access */
    if (class1 == class2)
        return true;

    /* class loaders must match */
    if (class1->classLoader != class2->classLoader) //classLoader不一致，直接
    return false
        return false;

    /*
     * Switch array classes to their element types.  Arrays receive the
     * class loader of the underlying element type.  The point of doing
     * this is to get the un-decorated class name, without all the
```

```
     * "[[L...;" stuff.
     */
    if (dvmIsArrayClass(class1))
        class1 = class1->elementClass;
    if (dvmIsArrayClass(class2))
        class2 = class2->elementClass;

    /* check again */
    if (class1 == class2)
        return true;

    /*
     * We have two classes with different names.  Compare them and see
     * if they match up through the final '/'.
     *
     *  Ljava/lang/Object; + Ljava/lang/Class;          --> true
     *  LFoo;              + LBar;                      --> true
     *  Ljava/lang/Object; + Ljava/io/File;             --> false
     *  Ljava/lang/Object; + Ljava/lang/reflect/Method; --> false
     */
    int commonLen;

    commonLen = strcmpCount(class1->descriptor, class2->descriptor);
    if (strchr(class1->descriptor + commonLen, '/') != NULL ||
        strchr(class2->descriptor + commonLen, '/') != NULL)
    {
        return false;
    }

    return true;
}
```

我们可以看到如果在当前类和实现接口或父类是非 public 时，同时负责加载两者的 classLoader 不一样的情况下，直接 return false。所以如果此时不进行任何处理，会在类加载阶段报错。我们当前的代码热修复方案是基于新 classLoader 加载补丁类，所以在打包的过程中就会报类似如下的错误。

```
Superclass of '%s' (%s) is not accessible
```

（2）类校验阶段访问权限检查

上一小节我们可以得出结论，在补丁加载阶段，就能得到异常，所以完全可以在

补丁加载失败的情况下，走冷部署方案。事实上并非如此简单，如果补丁类中存在非 public 类的访问或非 public 方法或域的调用，那么都会导致失败。更致命的是，在补丁加载阶段是检测不出来的，补丁会被视为正常加载，但是在运行阶段会直接 crash 异常退出。接下来我们深入分析。

由于补丁类在单独的 dex 文件中，所以如果要加载这个 dex 的话，肯定要进行 dexopt 的，dexopt 过程中会执行 dvmVerifyClass 校验 dex 中的每个类。方法调用链：dvmVerifyClass 校验类 → verifyMethod 校验类中的每个方法 → (dvmVerifyCodeFlow → doCodeVerification) 对每个方法的逻辑进行校验 → verifyInstruction 实际上就是校验每个指令：

```
static bool verifyInstruction(const Method* meth, InsnFlags* insnFlags,
    RegisterTable* regTable, int insnIdx, UninitInstanceMap* uninitMap,
    int* pStartGuess){
    ... ...
    case OP_NEW_INSTANCE:
        resClass = dvmOptResolveClass(meth->clazz, decInsn.vB, &failure);

    case OP_INVOKE_VIRTUAL:
    case OP_INVOKE_SUPER:
    case OP_INVOKE_DIRECT:
    case OP_INVOKE_STATIC:
            calledMethod = verifyInvocationArgs(meth, workLine, insnRegCount,
                           &decInsn, uninitMap, METHOD_VIRTUAL, isRange,
                           isSuper, &failure);
            if (!VERIFY_OK(failure))
                break
    ... ...
    if (!VERIFY_OK(failure)) {
        if (failure == VERIFY_ERROR_GENERIC || gDvm.optimizing) { /*失败的错误
        原因只要不是 VERIFY_ERROR_GENERIC 就会执行 else 分支 */
            /* immediate failure, reject class */
            LOG_VFY_METH(meth, "VFY:  rejecting opcode 0x%02x at 0x%04x",
                decInsn.opcode, insnIdx);
            goto bail;
        } else {
            /* replace opcode and continue on */
            ALOGD("VFY: replacing opcode 0x%02x at 0x%04x",
                decInsn.opcode, insnIdx);
            if (!replaceFailingInstruction(meth, insnFlags, insnIdx,
               failure)) { //指令替换
                LOG_VFY_METH(meth, "VFY:  rejecting opcode 0x%02x at 0x%04x",
```

```
                    decInsn.opcode, insnIdx);
                goto bail;
            }
            /* IMPORTANT: meth->insns may have been changed */
            insns = meth->insns + insnIdx;

            /* continue on as if we just handled a throw-verification-error
            */
            failure = VERIFY_ERROR_NONE;
            nextFlags = kInstrCanThrow;
        }
    }
```

OP_NEW_INSTANCE 指令的校验会通过 dvmOptResolveClass → dvmCheckClassAccess 去检查补丁类所引用的类的访问权限，dvmCheckClassAccess 这个方法前面介绍过，不重复介绍，我们着重介绍下 OP_INVOKE_VIRTUAL 指令的校验。上面可以看到调用的是 verifyInvocationArgs → dvmResolveMethod。

```
/*
 * Alternate version of dvmResolveMethod().
 *
 * Doesn't throw exceptions, and checks access on every lookup.
 *
 * On failure, returns NULL, and sets *pFailure if pFailure is not NULL.
 */
Method* dvmOptResolveMethod(ClassObject* referrer, u4 methodIdx,
    MethodType methodType, VerifyError* pFailure)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    Method* resMethod;

    assert(methodType == METHOD_DIRECT ||
           methodType == METHOD_VIRTUAL ||
           methodType == METHOD_STATIC);

    resMethod = dvmDexGetResolvedMethod(pDvmDex, methodIdx);
    if (resMethod == NULL) {
        const DexMethodId* pMethodId;
        ClassObject* resClass;

        pMethodId = dexGetMethodId(pDvmDex->pDexFile, methodIdx);

        resClass = dvmOptResolveClass(referrer, pMethodId->classIdx,
                       pFailure);
        ......
```

```
    }

    bool allowed = dvmCheckMethodAccess(referrer, resMethod); // 方法的访问权限检查
    untweakLoader(referrer, resMethod->clazz);
    if (!allowed) {
        IF_ALOGI() {
            char* desc = dexProtoCopyMethodDescriptor(&resMethod->prototype);
            ALOGI("DexOpt: illegal method access (call %s.%s %s from %s)",
                resMethod->clazz->descriptor, resMethod->name, desc,
                referrer->descriptor);
            free(desc);
        }
        if (pFailure != NULL)
            *pFailure = VERIFY_ERROR_ACCESS_METHOD;
        return NULL;
    }

    return resMethod;
}
```

我们看到此时调用 dvmCheckMethodAccess 检查调用方法的访问权限。

```
bool dvmCheckMethodAccess(const ClassObject* accessFrom, const Method*
method){
    return checkAccess(accessFrom, method->clazz, method->accessFlags);
}
/*
 * Validate method/field access.
 */
static bool checkAccess(const ClassObject* accessFrom,
    const ClassObject* accessTo, u4 accessFlags){
    /* quick accept for public access */
    if (accessFlags & ACC_PUBLIC)
        return true;

    /* quick accept for access from same class */
    if (accessFrom == accessTo)
        return true;

    /* quick reject for private access from another class */
    if (accessFlags & ACC_PRIVATE)
        return false;

    /*
     * Semi-quick test for protected access from a sub-class, which may or
     * may not be in the same package.
     */
    if (accessFlags & ACC_PROTECTED)
```

```
        if (dvmIsSubClass(accessFrom, accessTo))
            return true;

    /*
     * Allow protected and private access from other classes in the same
     * package.
     */
    return dvmInSamePackage(accessFrom, accessTo);
}
```

如果该方法的属性是 public, checkAccess 直接返回 true；如果是 protected, 那么检查调用方法所在的类和当前类是否是父子类的关系，如果不是父子类则调用 dvmInSamePackage, 这个方法前面已经介绍过；如果补丁类和方法所在的类不是 classLoader, 那么返回 false。

所以最后能够得出结论，补丁类如果引用了非 public 类，那么 verifyInstruction 方法执行的结果 `pFailure = VERIFY_ERROR_ACCESS_METHOD`，verifyInstruction 方法接下来就会执行 replaceFailingInstruction 指令替换流程。replaceFailingInstruction → dvmUpdateCodeUnit 方法更新**潜在错误的 opcode 指令码**，此时会把上述权限检查失败的指令码比如 invoke-virtual/new-instance 等替换成为 `OP_THROW_VERIFICATION_ERROR` 指令。

```
static bool replaceFailingInstruction(const Method* meth, InsnFlags*
insnFlags,
    int insnIdx, VerifyError failure)
{
    VerifyErrorRefType refType;
    u2* oldInsns = (u2*) meth->insns + insnIdx;
    int width;

    Opcode opcode = dexOpcodeFromCodeUnit(*oldInsns);
    switch (opcode) {

    case OP_NEW_INSTANCE:
        refType = VERIFY_ERROR_REF_CLASS;
        break;

    case OP_IGET:                       // insn[1] == field ref, 2 bytes
        refType = VERIFY_ERROR_REF_FIELD;
        break;
```

```
    case OP_INVOKE_VIRTUAL:              // insn[1] == method ref, 3 bytes
    case OP_INVOKE_SUPER:
    case OP_INVOKE_DIRECT:
    case OP_INVOKE_STATIC:
    case OP_INVOKE_INTERFACE:
        refType = VERIFY_ERROR_REF_METHOD;
        break;
    }

    /* encode the opcode, with the failure code in the high byte */
    assert(width == 2 || width == 3);
    u2 newVal = OP_THROW_VERIFICATION_ERROR |
        (failure << 8) | (refType << (8 + kVerifyErrorRefTypeShift));
    dvmUpdateCodeUnit(meth, oldInsns, newVal);
    return true;
}
```

注意：此时 verifyInstruction 返回的仍然是 true，所以 dvmVerifyClass 返回的也是 true，所以在补丁加载的时候是成功的，我们并不会收到任何异常。这也是在跟补丁类的父类或实现的接口非 public 情况下进行补丁加载就报异常最大的区别。那么接下来看下 `OP_THROW_VERIFICATION_ERROR` 指令会发生什么：

```
HANDLE_OPCODE(OP_THROW_VERIFICATION_ERROR)
    EXPORT_PC();
    vsrc1 = INST_AA(inst);
    ref = FETCH(1);             /* class/field/method ref */
    dvmThrowVerificationError(curMethod, vsrc1, ref); /*调用
    dvmThrowVerificationError方法*/
    GOTO_exceptionThrown();
OP_END

void dvmThrowVerificationError(const Method* method, int kind, int ref){
    switch ((VerifyError) errorKind) {
    case VERIFY_ERROR_ACCESS_CLASS:
        exceptionClass = gDvm.exIllegalAccessError;
        msg = classNameFromIndex(method, ref, refType,
            kThrowShow_accessFromClass); //类权限异常信息
        break;
    case VERIFY_ERROR_ACCESS_FIELD:
        exceptionClass = gDvm.exIllegalAccessError;
        msg = fieldNameFromIndex(method, ref, refType,
            kThrowShow_accessFromClass); //域权限异常信息
        break;
```

```
    case VERIFY_ERROR_ACCESS_METHOD:
        exceptionClass = gDvm.exIllegalAccessError;
        msg = methodNameFromIndex(method, ref, refType,
            kThrowShow_accessFromClass); // 方法权限异常信息
        break;
    ... ...
    }

    dvmThrowException(exceptionClass, msg.c_str());
}
```

所以，我们知道了在执行的过程中 `dvmThrowException` 抛出异常，程序中断异常退出。

2.3.10 <clinit>方法

由于补丁热部署模式的特殊性——不允许类结构变更以及不允许变更 `<clinit>` 方法，所以补丁工具如果发现了这几种限制情况，那么此时只能走冷启动重启生效，冷启动几乎是无任何限制的，可以做到任何场景的修复。

可能有时候在源码层上来看并没有新增或减少 method 和 field，但是实际上由于要满足 Java 各种语法特性的需求，所以编译器会在编译期间自动合成一些 method 和 field，最后就有可能触发了这几个限制情况。以上列举的情况可能并不完全详细，这些分析也只是起抛砖引玉的作用，具体情况还需要具体分析，同时一些难以理解的 Java 语法特性或许从编译的角度去分析可能显而易见了。

仅供非商业用途或交流学习使用

2.4 本章小结

本章详细介绍了热替换热修复的实现原理，并重点讲解了影响热替换热修复的一些重要的编译期问题。总体而言，热替换方式的修复无须重启可以直接生效，这种立竿见影的修复效果也是它的最大优势。然而，除了我们之前描述的技术上的限制，热替换修复还有个比较大的问题。由于是在应用运行期间发生了变动，如果我们修改了某个方法的逻辑，就会导致它在修复前后的逻辑不一致，这就会引发一些诡异的错误。因此，热替换方式的热修复只适用于修复一些简单的 BUG，如果要做一些功能方面的更新，不建议采用。

第 3 章
冷启动代码修复

对冷启动修复技术进行了深入的剖析。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

3.1 冷启动类加载原理

前面我们提到热部署修复方案有诸多特点（有关热替换方案的实现请看第 2 章）。其根本原理是基于 native 层方法的替换，所以当类结构变化时，如新增减少类 method/field 在热部署模式下会受到限制。但冷部署能突破这种约束，可以更好地达到修复目的，再加上冷部署在稳定性上具有的独特优势，因此可以作为热部署的有力补充而存在。

3.1.1 冷启动实现方案概述

冷启动重启生效，现在一般有两种实现方案（如表 3-1 所示），同时给出它们各自的优缺点。

表 3-1 冷启动的实现方案

	QQ空间	Tinker
原理	为了解决 Dalvik 下 unexpected dex problem 异常而采用插桩的方式，单独放一个帮助类在独立的 dex 中让其他类调用，阻止了类被打上 CLASS_ISPREVERIFIED 标志从而规避问题的出现。最后加载补丁 dex 得到 dexFile 对象作为参数构建一个 Element 对象插入到 dex-Elements 数组的最前面	提供 dex 差量包，整体替换 dex 的方案。差量的方式给出 patch.dex，然后将 patch.dex 与应用的 classes.dex 合并成一个完整的 dex，完整 dex 加载得到的 dexFile 对象作为参数构建一个 Element 对象然后整体替换掉旧的 dex-Elements 数组

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

（续表）

	QQ空间	Tinker
优点	没有合成整包，产物比较小，比较灵活	自研 dex 差异算法，补丁包很小，dex merge 成完整 dex，Dalvik 不影响类加载性能，Art 下也不存在必须包含父类 / 引用类的情况
缺点	Dalvik 下影响类加载性能，Art 下类地址写死，导致必须包含父类 / 引用类，最后补丁包很大	dex 合并内存消耗在 vm heap 上，容易 OOM，最后导致 dex 合并失败

通过表 3-1，我们能清晰地看到两种方案的缺点都很明显。这里对 tinker 方案 dex merge 缺陷进行简单说明。

dex merge 操作是在 Java 层面进行的，所有对象的分配都是在 java heap 上完成的，如果此时进程申请的 java heap 对象超过了 `vm heap` 规定的大小，那么进程发生 OOM，系统 memory killer 可能会杀掉该进程，导致 dex 合成失败。另外一方面，我们知道 JNI 层面 `C++ new/malloc` 申请的内存，分配在 `native heap`，native heap 的增长并不受 vm heap 大小的限制，只受限于 RAM，如果 RAM 不足，那么进程也会被杀死导致闪退。所以如果只是从 dex merge 方面思考，在 JNI 层面进行 dex merge，从而可以避免 OOM，提高 dex 合并的成功率。理论上当然可以，只是 JNI 层实现起来比较复杂而已

前文说过，我们的需求是冷启动模式是热部署模式的补充方案，所以这两种方案使用的应该是同一套补丁，另外一方面，跟代码修复热部署方案一样，我们追求的是不侵入打包。上述两种方案都需要侵入应用打包，同时补丁的结构也不一样，这两套方案对我们来说都不适用。所以我们需要另辟蹊径，寻求一种既能无侵入打包又能做热部署模式的补充解决方案，下面将分别对 Dalvik 虚拟机和 Art 虚拟机的冷启动方案分别进行介绍。

3.1.2 插桩实现的前因后果

众所周知，如果仅仅把补丁类打入补丁包中而不做任何处理的话，那么在运行时

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

类加载的时候就会产生异常并退出，接下来先来看一下抛出这个异常的前因后果。

加载一个 dex 文件到本地内存的时候，如果不存在 odex 文件，那么首先会执行 dexopt，dexopt 的入口在 `davilk/opt/OptMain.cpp` 的 main 方法中，最后调用到 `verifyAndOptimizeClass` 执行真正的 verify/optimize 操作。

```
/*
 * Verify and/or optimize a specific class.
 */
static void verifyAndOptimizeClass(DexFile* pDexFile, ClassObject* clazz,
        const DexClassDef* pClassDef, bool doVerify, bool doOpt) {
    const char* classDescriptor;
    bool verified = false;
    classDescriptor = dexStringByTypeIdx(pDexFile, pClassDef->classIdx);

    if (doVerify) {
        if (dvmVerifyClass(clazz)) { //执行类的 Verify
            ((DexClassDef*)pClassDef)->accessFlags |= CLASS_ISPREVERIFIED; /*
            类被打上 CLASS_ISPREVERIFIED 标志 */
            verified = true;
        }
    }

    if (doOpt) {
        bool needVerify = (gDvm.dexOptMode == OPTIMIZE_MODE_VERIFIED || gDvm.
dexOptMode == OPTIMIZE_MODE_FULL);
        if (!verified && needVerify) {
            ... ...
        } else {
            dvmOptimizeClass(clazz, false); //执行类的 Optimize
            ((DexClassDef*)pClassDef)->accessFlags |= CLASS_ISOPTIMIZED; /* 类
            被打上 CLASS_ISOPTIMIZED 标志 */
        }
    }
}
```

在第一次安装 APK 的时候，会对原 dex 执行 dexopt，此时假如 APK 只存在一个 dex，dvmVerifyClass(clazz) 返回结果为 true。然后 APK 中所有的类都会被打上 `CLASS_ISPREVERIFIED` 标志，接下来执行 dvmOptimizeClass，类接着被打上 `CLASS_ISOPTIMIZED` 标志。

- dvmVerifyClass: 类校验，简单来说，类校验的目的就是为了防止校验类的

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

合法性被篡改。此时会对类的每个方法进行校验，这里我们只需要知道如果类的所有方法中直接引用到的类（第一层级关系，不会进行递归搜索）和当前类都在同一个 dex 中的话，dvmVerifyClass 就返回 true。

- dvmOptimizeClass: 类优化，简单来说这个过程会把部分指令优化成虚拟机内部指令，比如方法调用指令 `invoke-*` 变成了 `invoke-*-quick`，quick 指令会从类的 vtable 表中直接获取，vtable 简单来说就是类的所有方法的一张大表（包括继承自父类的方法）。因此提升了方法的执行速率。

现在假定 A 类是补丁类，所以补丁 A 类在单独的 dex 中。类 B 中的某个方法引用到补丁类 A，所以执行到该方法会尝试解析类 A。

```
ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx, bool
fromUnverifiedConstant) {
    ... ...
    if (!fromUnverifiedConstant && IS_CLASS_FLAG_SET(referrer, CLASS_
ISPREVERIFIED)){ // 如果类被打上了 CLASS_ISPREVERIFIED 标志
        if (referrer->pDvmDex != resClassCheck->pDvmDex && resClassCheck-
            >classLoader != NULL) {
            dvmThrowIllegalAccessError("Class ref in pre-verified class
            resolved to unexpected implementation"); // 抛出异常
            return NULL;
        }
    }
    ... ...
}
```

通过上面的代码很容易看出，类 B 由于被打上了 CLASS_ISPREVERIFIED 标志，接下来 referrer 是类 B，resClassCheck 是补丁类 A，它们属于不同的 dex。所以抛出 dvmThrowIllegalAccessError 异常。为了解决这个问题，一个单独的无关帮助类被放到一个单独的 dex 中，原 dex 中所有类的构造函数都引用这个类，一般的实现方法都是侵入 dex 打包流程，利用 .class 字节码修改技术，在所有 .class 文件的构造函数中引用这个帮助类，插桩由此而来。根据前面的介绍，在 dexopt 过程中 dvmVerifyClass 类校验返回 false，原 dex 中所有的类都没有 CLASS_ISPREVERIFIED 标志，因此解决运行时的这个异常。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

但是插桩会给类加载效率带来比较严重的影响。熟悉 Dalvik 虚拟机的开发者知道，一个类的加载通常有三个阶段：dvmResolveClass、dvmLinkClass、dvmInitClass，本书对这三个阶段不再进行详细说明。dvmInitClass 阶段在类解析完并尝试初始化类的时候执行，这个方法主要完成父类的初始化、当前类的初始化、静态变量的初始化赋值等操作。

```
bool dvmInitClass(ClassObject* clazz) {
    if (clazz->status < CLASS_VERIFIED) { // 如果类没被打上 CLASS_ISPREVERIFIED 标志
        clazz->status = CLASS_VERIFYING;
        if (!dvmVerifyClass(clazz)) { // 类 Verify
            ... ...
        }
        clazz->status = CLASS_VERIFIED;
    }
    if (!IS_CLASS_FLAG_SET(clazz, CLASS_ISOPTIMIZED) && !gDvm.optimizing) {
       // 如果类没被打上 CLASS_ISOPTIMIZED 标志
        dvmOptimizeClass(clazz, essentialOnly);  // 类 Optimize
        SET_CLASS_FLAG(clazz, CLASS_ISOPTIMIZED);
    }
    ... ...
}
```

可以看到除了上面说过的类初始化之外，如果类没被打上 CLASS_ISPREVERIFIED/CLASS_ISOPTIMIZED 的标志，那么类的校验和优化都将在类的初始化阶段进行。正常情况下类的校验和优化都仅在 APK 第一次安装执行 dexopt 操作的时候进行，类的校验任务实际上是很重的，因为会对类的所有方法中的所有指令都进行校验，单个类加载时类校验耗时并不多，但是如果是在同一时间点加载大量类的情况下，这种耗时就会被放大。所以这也是插桩给类的加载效率带来比较大影响的后果，接下来来看一下具体会给类加载带来多大的影响。

3.1.3 插桩导致类加载性能影响

通过 3.1.2 节的介绍，我们知道若采用插桩，会导致所有类都非 preverify，从而导致校验与优化操作会在类加载时触发。这会导致类加载有一定的性能损耗。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

平均每个类校验与优化（跟类的大小有关系）的耗时并不长，而且这个耗时每个类只有一次（类只会加载一次）。但由于在应用刚启动时这种场景下一般会同时加载大量的类，因此在这种情况下影响还是比较大的，启动的时候就容易白屏，这一点用户是没法容忍的。

3.1.4 避免插桩的 QFix 方案

手 Q 热补丁轻量级 QFix 方案提供了一种不同的的思路。

```
ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx, bool
    fromUnverifiedConstant)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    ClassObject* resClass;
    const char* className;

    // 3. 提前把类设置进虚拟机后，可以直接返回 resClass
    resClass = dvmDexGetResolvedClass(pDvmDex, classIdx);
    if (resClass != NULL)
          return resClass;
    className = dexStringByTypeIdx(pDvmDex->pDexFile, classIdx);
    if (className[0] != '\0' && className[1] == '\0') {
        /* primitive type */
        resClass = dvmFindPrimitiveClass(className[0]);
    } else {
            resClass = dvmFindClassNoInit(className, referrer->classLoader);
    }
    // 1. 设置 fromUnverifiedConstant 为 true，不进入判断而绕过检查
    if (resClass != NULL) {
          if (!fromUnverifiedConstant &&
                IS_CLASS_FLAG_SET(referrer, CLASS_ISPREVERIFIED))
          {
                ClassObject* resClassCheck = resClass;
                if (dvmIsArrayClass(resClassCheck))
                      resClassCheck = resClassCheck->elementClass;
                if (referrer->pDvmDex != resClassCheck->pDvmDex &&
                      resClassCheck->classLoader != NULL)
                {
                      ALOGW("Class resolved by unexpected DEX:"
                          " %s(%p):%p ref [%s] %s(%p):%p",
                         referrer->descriptor, referrer->classLoader,
                         referrer->pDvmDex,
                         resClass->descriptor, resClassCheck->descriptor,
                         resClassCheck->classLoader, resClassCheck->pDvmDex);
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
                    ALOGW("(%s had used a different %s during pre-verification)",
                        referrer->descriptor, resClass->descriptor);
                    dvmThrowIllegalAccessError(
                        "Class ref in pre-verified class resolved to unexpected "
                        "implementation");
                    return NULL;
                }
            }

            LOGVV("##### +ResolveClass(%s): referrer=%s dex=%p ldr=%p ref=%d",
                resClass->descriptor, referrer->descriptor, referrer->pDvmDex,
                referrer->classLoader, classIdx);

            // 2. 顺利设置进虚拟机中
            dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);
        } else {
            /* not found, exception should be raised */
            LOGVV("Class not found: %s",
                dexStringByTypeIdx(pDvmDex->pDexFile, classIdx));
            assert(dvmCheckException(dvmThreadSelf()));
        }

        return resClass;
    }
```

怎么让 dvmDexGetResolvedClass 返回的结果不为 null？只要调用过一次 `dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);` 就可以了，下面举个例子简单说明。

```
public class B {
    public static void test() {
        A.a();
    }
}
```

我们此时需要打包的类是类 A，所以类 A 被打入到一个独立的补丁 dex 文件中。那么执行到类 B 的 test 方法时，A.a() 这行代码时就会尝试去解析类 A，此时进行 `dvmResolveClass(const ClassObject* referrer, u4 classIdx, bool fromUnverifiedConstant)`。

- referrer: 实际上就是类 B。
- classIdx: 类 A 在原 dex 文件结构类区中的索引 ID。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- fromUnverifiedConstant: 是否执行 const-class/instance-of 指令。

此时调用的是类 A 的静态 a 方法，`invoke-static` 指令不属于 `const-class/instance-of` 这两个指令。如果不做任何处理的话，dvmDexGetResolvedClass 的返回值是 null。因为类 A 是从补丁 dex 中解析加载的，类 B 是在原 dex 中，所以 `B->pDvmDex != A->pDvmDex`，接下来执行到 `dvmThrowIllegalAccessError` 从而导致运行时异常。为避免异常，必须要在开始的时候，就把补丁类 A 添加到原有 dex(pDvmDex) 的 pResClasses 数组中。这样就确保了在执行类 B 的 test 方法时，dvmDexGetResolvedClass 返回值不为 null，也就不会执行后续类 A 和类 B 的 dex 一致性校验了。

具体实现，首先通过补丁工具反编译 dex 为 smali 文件拿到以下文件。

- preResolveClz: 需要打包的类 A 的描述符，非必需，为了调试方便加上该参数而已，如 Lcom/taobao/patch/demo/A。
- refererClz: 需要打包的类 A 所在的 dex 文件的任何一个类描述符，**注意，这里不限定必须是引用补丁类 A 的某个类，实际上只要是同一个 dex 中的任何一个类都可以。所以我们直接拿原 dex 中的第一个类即可**。如 Landroid/support/annotation/AnimRes。
- classIdx: 需要打包的类 A 在原有 dex 文件中的类索引 ID。如 2425。

然后通过 dlopen 拿到 libdvm.so 库的句柄，通过 dlsym 拿到该 so 库的 `dvmResolveClass/dvmFindLoadedClass` 函数指针。首先需要预加载引用类 android/support/annotation/AnimRes，这样 dvmFindLoadedClass("android/support/annotation/AnimRes") 返回值才不为 null，然后 `dvmFindLoadedClass` 执行结果得到的 ClassObject 作为第一个参数执行 `dvmResolveClass(AnimRes, 2425, true)` 即可。

下面简单看一下 JNI 层代码的部分实现。实际上可以看到 `preResolveClz` 参数是非必需的。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
jboolean resolveColdPatchClasses(JNIEnv *env, jclass clz, jstring
preResolveClz, jstring refererClz,
                                 jlong classIdx, dexstuff_t *dexstuff) {
    LOGD("start resolveColdPatchClasses");
    ClassObject *refererObj = dexstuff->dvmFindLoadedClass_fnPtr(
            Jstring2CStr(env, refererClz));// 调用 dvmFindLoadedClass
    LOGD("referrer ClassObject: %s\n", refererObj->descriptor);
    if (strlen(refererObj->descriptor) == 0) {
        return JNI_FALSE;
    }

    ClassObject *resolveClass = dexstuff->dvmResolveClass_fnPtr(refererObj,
classIdx, true);// 调用 dvmResolveClass
    LOGD("classIdx ClassObject: %s\n", resolveClass->descriptor);
    if (strlen(resolveClass->descriptor) == 0) {
        return JNI_FALSE;
    }
    return JNI_TRUE;
}
```

与前面方案一采用的 native hook 方式不同，这个思路不会去 hook 某个系统方法，而是从 native 层直接调用，同时更不需要插桩。具体实现需要注意以下 3 点：

- dvmResolveClass 的第三个参数 fromUnverifiedConstant 必须为 true。
- 在 APK 多 dex 的情况下，dvmResolveClass 第一个参数 referrer 类必须跟需要打包的类在同一个 dex 中，但是它们两个类不需要存在任何引用关系，任何一个在同一个 dex 中的类作为 referrer 都可以。
- referrer 类必须提前加载。

然而，QFix 的方案有它独特的缺陷，由于是在 dexopt 后绕过的，dexopt 会改变原有的很多逻辑，许多 odex 层面的优化会固定字段和方法的访问偏移，这就会导致比较严重的 BUG，3.2 节会详细解释这一影响。最后采用的是自研的全量 dex 方案，具体可看后面的 3.3 节。

3.1.5 Art 下冷启动实现

前面说过补丁在热部署模式下是一个完整的类，补丁的粒度是类。现在的需求是补丁既能走热部署模式也能走冷启动模式，**为了减小补丁包的大小，并没有为热部署**

仅供非商业用途或交流学习使用

和冷启动分别准备一套补丁，而是在同一个热部署模式下的补丁能够降级直接走冷启动，所以不需要做 dex merge。但是通过前面的阅读我们知道为了解决 Art 下类地址写死的问题，Tinker 通过 dex merge 成一个全新完整的新 dex 整体替换掉旧的 dexElements 数组。事实上并不需要这样做，Art 虚拟机下面默认已经支持多 dex 压缩文件的加载。

下面分别来看一下 Dalvik 和 Art 对 `DexFile.loadDex` 尝试把一个 dex 文件解析加载到 native 中内存都发生了什么。实际上都是调用了 `DexFile.openDexFileNative` 这个 native 方法。看一下 native 层对应的 C/C++ 代码具体实现。

在 Dalvik 虚拟机下实现：

```
static void Dalvik_dalvik_system_DexFile_openDexFileNative(const u4* args,
JValue* pResult) {
    if (hasDexExtension(sourceName)
            && dvmRawDexFileOpen(sourceName, outputName, &pRawDexFile, false)
               == 0) { // 加载一个原始 dex 文件
        ALOGV("Opening DEX file '%s' (DEX)", sourceName);

        pDexOrJar = (DexOrJar*) malloc(sizeof(DexOrJar));
        pDexOrJar->isDex = true;
        pDexOrJar->pRawDexFile = pRawDexFile;
        pDexOrJar->pDexMemory = NULL;
    } else if (dvmJarFileOpen(sourceName, outputName, &pJarFile, false) == 0)
    { // 加载一个压缩文件
        ALOGV("Opening DEX file '%s' (Jar)", sourceName);

        pDexOrJar = (DexOrJar*) malloc(sizeof(DexOrJar));
        pDexOrJar->isDex = false;
        pDexOrJar->pJarFile = pJarFile;
        pDexOrJar->pDexMemory = NULL;
    } else {
        ALOGV("Unable to open DEX file '%s'", sourceName);
        dvmThrowIOException("unable to open DEX file");
    }
}

int dvmJarFileOpen(const char* fileName, const char* odexOutputName,
JarFile** ppJarFile, bool isBootstrap){
    ... ...
    else {
        ZipEntry entry;
tryArchive:
        /*
```

仅供非商业用途或交流学习使用


```
            * Pre-created .odex absent or stale。 Look inside the jar for a
            * "classes.dex".
            */
           entry = dexZipFindEntry(&archive, kDexInJarName);/*
              kDexInJarName=="classes.dex"，说明只加载一个 dex*/
           ... ...
       }
   }
```

通过 `static const char* kDexInJarName = "classes.dex";` 可以发现，很明显 Dalvik 尝试加载一个压缩文件的时候只会去把 `classes.dex` 加载到内存中。如果此时压缩文件中有多个 dex，那么**除了 `classes.dex` 之外的其他 dex 被直接忽略掉。**

在 Art 虚拟机下：方法调用链 `DexFile_openDexFileNative-> OpenDexFilesFromOat -> LoadDexFiles`。

```
std::vector<std::unique_ptr<const DexFile>> OatFileAssistant::LoadDexFiles(
    const OatFile& oat_file, const char* dex_location) {

  // Load the primary dex file.
  const OatFile::OatDexFile* oat_dex_file = oat_file.GetOatDexFile(
                                            dex_location, nullptr,
                                            false);
  std::unique_ptr<const DexFile> dex_file = oat_dex_file->OpenDexFile(&error_
                                            msg);
  dex_files.push_back(std::move(dex_file));

  // Load secondary multidex files
  for (size_t i = 1; ; i++) {
    std::string secondary_dex_location = DexFile::GetMultiDexLocation(i, dex_
                                         location);
    oat_dex_file = oat_file.GetOatDexFile(secondary_dex_location.c_str(),
                        nullptr, false);
    dex_file = oat_dex_file->OpenDexFile(&error_msg);
    dex_files.push_back(std::move(dex_file));
  }
  return dex_files;
}
```

从上面的代码中大概可以看出，在 Art 下默认已经支持加载压缩文件中包含多个 dex，首先肯定优先加载 primary dex 也就是 `classes.dex`，后续会加载其他的 dex。所以补丁类只需要放到 `classes.dex` 中即可，后续出现在其他 dex 中

的“补丁类”是不会被重复加载的。所以我们得到在 Art 下最终的冷启动解决方案：**我们只要把补丁 dex 命名为 classes.dex。原 APK 中的 dex 依次命名为 classes(2,3,4...).dex 就可以了，然后一起打包为一个压缩文件。再通过 DexFile.loadDex 得到 DexFile 对象，最后用该 DexFile 对象整体替换旧的 dexElements 数组就可以了。**

我们的方案和 Tinker 方案的不同点如图 3-1 所示。

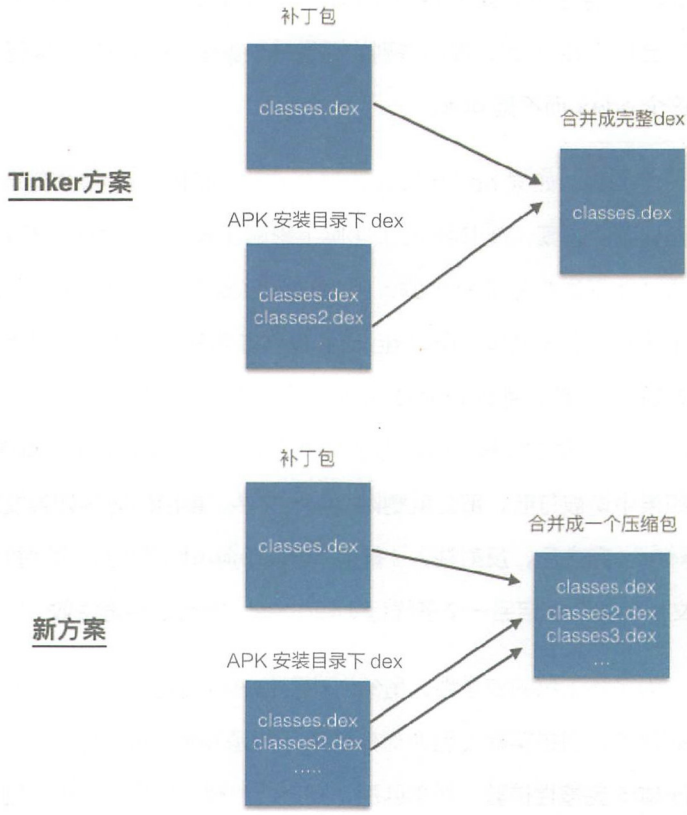


图 3-1 方案不同点

需要注意：

- 补丁 dex 必须命名为 `classes.dex`；
- 用 loadDex 得到的 DexFile 完整替换掉 dexElements 数组而不是插入。

3.1.6 不得不说的其他点

我们知道 `DexFile.loadDex` 尝试把一个 dex 文件解析并加载到 native 内存中，在加载到 native 内存之前，如果 dex 不存在对应的 odex，那么 Dalvik 下会执行 dexopt，Art 下会执行 dexoat，最后得到的都是一个优化后的 odex。**实际上最后虚拟机执行的是这个 `odex` 而不是 `dex`。**

现在有这样一个问题，如果 dex 足够大，那么 dexopt/dexoat 操作实际上是很耗时的，根据上面提到的方案，在 Dalvik 下实际上影响比较小，因为 loadDex 仅仅是补丁包。但是在 Art 下影响是非常大的，因为 loadDex 是补丁 dex 和 APK 中原 dex 合并成的一个完整补丁压缩包，所以 dexoat 操作非常耗时。所以如果优化后的 odex 文件没生成或者不完整，那么 loadDex 便不能在应用启动的时候进行，因为会阻塞 loadDex 线程，一般是主线程。所以为了解决这个问题，**我们把 `loadDex` 当作一个事务来看，如果中途被打断，那么就删除 `odex` 文件，重启的时候如果发现存在 `odex` 文件，`loadDex` 完之后，反射注入 / 替换 `dexElements` 数组，实现打包。如果不存在 `odex` 文件，那么重启另一个子线程 `loadDex`，重启之后再生效。**

另外一方面，为了补丁包的安全性，虽然对补丁包进行签名校验，这个时候能够防止整个补丁包被篡改，但是实际上因为虚拟机执行的是 odex 而不是 dex，还需要对 **`odex` 文件进行 `MD5` 完整性校验**，如果匹配，则直接加载，如果不匹配，则重新生成 odex 文件，防止 odex 文件被篡改。

3.1.7 完整的方案考虑

代码修复冷启动方案由于其高兼容性，几乎可以修复任何场景下的代码缺陷，但

是注入前被加载的类（比如 Application 类）肯定是不能被修复的。所以我们把它作为一个保底的方案，在没法应用热部署或者热部署失败的情况下，最后都会应用代码冷启动重启生效方案，所以我们的补丁是同一套的。具体实施方案对 Dalvik 下和 Art 下分别做了处理：

- 在 Dalvik 下采用自行研发的全量 dex 方案。
- 在 Art 下本质上虚拟机已经支持多 dex 的加载，我们要做的仅仅是把补丁 dex 作为主 dex(classes.dex) 加载而已。

3.2 多态对冷启动类加载的影响

前面我们知道冷启动方案几乎可以修复任何场景的代码缺陷，但 Dalvik 下的 QFix 方案存在很大的限制，下面将深入介绍在目前方案下为什么会有这些限制，同时给出具体的解决方案。

3.2.1 重新认识多态

实现多态的技术一般叫作动态绑定，是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。多态一般指的是非静态非私有方法的多态，field 和静态方法不具有多态性。示例如下。

```
public class B extends A {
    String name = "B name";

    @Override
    void a_t1() {
        System.out.println("B a_t1...");
    }
    void b_t1(){}

    public static void main(String[] args) {
        A obj = new B();
        System.out.println(obj.name);
        obj.a_t1();
```

```
    }
}
class A {
    String name = "A name";

    void a_t1() {
        System.out.println("A a_t1...");
    }
    void a_t2(){}
}
```

输出结果：`A name/B a_t1...`，可以看到 name 这个 field 没有多态性，print 这个方法具有多态性，这里先分析一下方法多态性的实现。首先 `new B()` 的执行会尝试加载类 B，方法调用链 `dvmResolveClass->dvmLinkClass->creatEVtable`，此时会为类 B 创建一个 vtable，**其实在虚拟机中加载每个类都会为这个类生成一张 `vtable` 表，`vtable` 表就是当前类的所有 `virtual` 方法的一个数组，当前类和所有继承父类的 `public/protected/default` 方法就是 `virtual` 方法，因为 `public/protected/default` 修饰的方法是可以被继承的。`private/static` 方法不属于这个范畴，因为不能被继承。**

```
/*
 * Create the virtual method table.
 *
 * The top part of the table is a copy of the table from our superclass,
 * with our local methods overriding theirs。 The bottom part of the table
 * has any new methods we defined.
 */
static bool createVtable(ClassObject* clazz){
    bool result = false;
    int maxCount;
    int i;

    /* the virtual methods we define, plus the superclass vtable size */
    maxCount = clazz->virtualMethodCount;
    if (clazz->super != NULL) {
        maxCount += clazz->super->vtableCount;/* 如果父类不为 null，那么当前类的
        vtable 表的大小就是当前类的 virtual 方法总数加上父类的 vtableCount*/
    }
    /*
     * Over-allocate the table, then realloc it down if necessary。 So
     * long as we don't allocate anything in between we won't cause
     * fragmentation, and reducing the size should be unlikely to cause
```

```
     * a buffer copy.
     */
    dvmLinearReadWrite(clazz->classLoader, clazz->virtualMethods);
    clazz->vtable = (Method**) dvmLinearAlloc(clazz->classLoader,
    sizeof(Method*) * maxCount); // 内存分配，vtable 的大小为 maxCount

    if (clazz->super != NULL) { // 存在父类
        int actualCount;

        memcpy(clazz->vtable, clazz->super->vtable,
            sizeof(*(clazz->vtable)) * clazz->super->vtableCount); /* 很关键的
一步，把父类所有的 vtable 都复制给子类。所有子类的 vtable 中存在所有继承自父类的方法 */
        actualCount = clazz->super->vtableCount;

        /*
         * See if any of our virtual methods override the superclass.
         */
        for (i = 0; i < clazz->virtualMethodCount; i++) { /* 遍历子类自身所有的
            virtual 方法 */
            Method* localMeth = &clazz->virtualMethods[i];
            int si;

            for (si = 0; si < clazz->super->vtableCount; si++) {
                Method* superMeth = clazz->vtable[si];

                if (dvmCompareMethodNamesAndProtos(localMeth, superMeth) ==
0) { /* 如果子类 virtual 方法签名和父类一样，说明该方法被覆盖了。子类重写方法覆盖掉 vtable
        中父类的方法 */
                    ... ...
                    clazz->vtable[si] = localMeth; /* 所以 vtable 中 si 的索引必须
                    换成子类覆盖的方法 */
                    localMeth->methodIndex = (u2) si; /*methodIndex 也就是方法在
                    vtable 中的索引值 */
                    break;
                }
            }

            if (si == clazz->super->vtableCount) { /* 方法签名和父类的不一致，说明
               是子类自己的方法 */
                /* not an override, add to end */
                clazz->vtable[actualCount] = localMeth; // 添加到子类 vtable 的末尾
                localMeth->methodIndex = (u2) actualCount;
                actualCount++;
            }
        }
        ... ...
    } else { // 没有继承自任何类
        /* java/lang/Object case */
        int count = clazz->virtualMethodCount;
```

```
        if (count != (u2) count) {
            ALOGE("Too many methods (%d) in base class '%s'", count,
                clazz->descriptor);
            goto bail;
        }

        for (i = 0; i < count; i++) {
            clazz->vtable[i] = &clazz->virtualMethods[i];
            clazz->virtualMethods[i].methodIndex = (u2) i;
        }
        clazz->vtableCount = clazz->virtualMethodCount;
    }
}
```

上面的代码注释解释得很清楚，子类 vtable 的大小等于子类 virtual 方法数 + 父类 vtable 的大小。

- 整体复制父类 vtable 到子类的 vtable；
- 遍历子类的 virtual 方法集合，如果方法原型一致，说明是重写父类方法，那么在相同索引位置处，子类重写方法覆盖掉 vtable 中父类的方法；
- 若方法原型不一致，那么把该方法添加到 vtable 的末尾。

所以在上述示例中，假如父类 A 的 vtable 是 `vtable[0]=A.a_t1, vtable[1]=A.a_t2` 方法，那么类 B 的 vtable 就是 `vtable[0]=B.a_t1, vtable[1]=A.a_t2, vtable[2]=B.b_t1`。接下来看一下 `obj.a_t1()` 发生了什么。invoke-Virtual 指令的解释如下：

```
GOTO_TARGET(invokeVirtual, bool methodCallRange, bool) {
    Method* baseMethod;
    Object* thisPtr;

    EXPORT_PC();

    vsrc1 = INST_AA(inst);      /* AA (count) or BA (count + arg 5) */
    ref = FETCH(1);             /* method ref */
    vdst = FETCH(2);            /* 4 regs -or- first reg */

    /*
     * The object against which we are executing a method is always
     * in the first argument.
     */
```

```
    if (methodCallRange) {
        thisPtr = (Object*) GET_REGISTER(vdst);
    } else {
        thisPtr = (Object*) GET_REGISTER(vdst & 0x0f); //当前对象
    }

    /*
     * Resolve the method。 This is the correct method for the static
     * type of the object。 We also verify access permissions here.
     */
    baseMethod = dvmDexGetResolvedMethod(methodClassDex, ref); /*是否已经解析过
该方法*/
    if (baseMethod == NULL) {
        baseMethod = dvmResolveMethod(curMethod->clazz, ref,METHOD_VIRTUAL);
        //没有解析过该方法调用 dvmResolveMethod, baseMethod 得到的当然是 A.a_t1 方法
        if (baseMethod == NULL) {
            ILOGV("+ unknown method or access denied");
            GOTO_exceptionThrown();
        }
    }

    /*
     * Combine the object we found with the vtable offset in the
     * method.
     */
    assert(baseMethod->methodIndex < thisPtr->clazz->vtableCount);
    methodToCall = thisPtr->clazz->vtable[baseMethod->methodIndex]; /*A.a_t1
    方法在类 A 的 vtable 中的索引去类 B 的 vtable 中查找*/

    ... ...

    GOTO_invokeMethod(methodCallRange, methodToCall, vsrc1, vdst);
}
GOTO_TARGET_END
```

首先 obj **引用类型**是基类 A，所以上述代码中 `baseMethod` 拿到的 `A.a_t1`，`baseMethod->methodIndex` 是该方法在类 A 的 vtable 中的索引 0，obj 的**实际类型**是类 B，所以 `thisPtr->clazz` 就是类 B，那么 `B.vtable[0]` 就是 `B.a_t1` 方法，所以 `obj.a_t1()` 实际最后调用的是 **`B.a_t1`** 方法。这样就实现了方法的多态。

至于 field/static 方法为什么不具有多态性，这里不进行详细的代码分析，有需要的读者可以看 `iget/invoke-static` 指令的解释，简单来讲，**是从当前变量的**

引用类型而不是实际类型中查找，如果找不到，再去父类中递归查找。所以 field 和 static 方法不具备多态性。

3.2.2 冷启动方案限制

下面来看一下如果新增了一个 public/protected/default 方法，会出现什么情况。

```
public class Demo {
    public static void test_addMethod() {
        A obj = new A();
        obj.a_t2();
    }
}
class A {
    int a = 0;

    //新增 a_t1 方法
    void a_t1() {
        Log.d("Sophix", "A a_t1");
    }
    void a_t2() {
        Log.d("Sophix", "A a_t2");
    }
}
```

修复后的 APK 中新增了 `a_t1()` 方法，Demo 类不做任何修复，测试发现应用补丁后 Demo.test_addMethod() 得到的结果竟然是 D/Sophix: A a_t1，这表明 `obj.a_t2` 执行的竟然是 a_t1 方法，简直匪夷所思。下面深入分析一下本质原因。

在 3.1 节提到过，在 dex 文件第一次加载的时候，会执行 dexopt，dexopt 有 verify 和 optimize 两个过程。

- dvmVerifyClass: 类校验，简单来说，类校验的目的就是为了防止类被篡改而校验类的合法性。此时会对类的每个方法进行校验，这里我们只需要知道如果类的所有方法中直接引用到的类（第一层级关系，不会进行递归搜索）和当前类都在同一个 dex 中的话，dvmVerifyClass 就返回 true。
- dvmOptimizeClass: 类优化，简单来说，这个过程会把部分指令优化成虚

拟机的内部指令，比如方法调用指令 invoke-virtual-quick，quick 指令会从类的 vtable 表中直接获取，vtable 简单来说就是类的所有方法的一张大表（包括继承自父类的方法）。因此提升了方法的执行速率。

这里主要介绍一下 optimize 阶段：

```
void dvmOptimizeClass(ClassObject* clazz, bool essentialOnly){
    int i;
    for (i = 0; i < clazz->directMethodCount; i++) {
        optimizeMethod(&clazz->directMethods[i], essentialOnly);
    }
    for (i = 0; i < clazz->virtualMethodCount; i++) {
        optimizeMethod(&clazz->virtualMethods[i], essentialOnly);
    }
}
static void optimizeMethod(Method* method, bool essentialOnly) { //优化类的方法
    ... ...
        /*
         * non-essential substitutions:
         *  invoke-{virtual,direct,static}[/range] --> execute-inline
         *  invoke-{virtual,super}[/range] --> invoke-*-quick
         */
        if (!matched && !essentialOnly) {
            switch (opc) {
            case OP_INVOKE_VIRTUAL:
                if (!rewriteExecuteInline(method, insns, METHOD_VIRTUAL)) {
                    rewriteVirtualInvoke(method, insns, OP_INVOKE_VIRTUAL_
QUICK); //重写 invoke-virtual 为虚拟机内部指令 invoke-virtual-quick
                }
                break;
            ... ...
        }
    ... ...
}
```

注释已经很清楚了，重写 `invoke-virtual` 为虚拟机内部指令 `invoke-virtual-quick`，这个指令后面跟的**立即数就是该方法在类 `vtable` 中的索引值。**

```
GOTO_TARGET(invokeVirtualQuick, bool methodCallRange){
    Object* thisPtr;
    EXPORT_PC();
    vsrc1 = INST_AA(inst);      /* AA (count) or BA (count + arg 5) */
    ref = FETCH(1);             /* vtable index */
    vdst = FETCH(2);            /* 4 regs -or- first reg */
```

```
    /*
     * The object against which we are executing a method is always
     * in the first argument.
     */
    if (methodCallRange) {
        assert(vsrc1 > 0);
        ILOGV("|invoke-virtual-quick-range args=%d @0x%04x {regs=v%d-v%d}",
            vsrc1, ref, vdst, vdst+vsrc1-1);
        thisPtr = (Object*) GET_REGISTER(vdst);
    } else {
        assert((vsrc1>>4) > 0);
        ILOGV("|invoke-virtual-quick args=%d @0x%04x {regs=0x%04x %x}",
            vsrc1 >> 4, ref, vdst, vsrc1 & 0x0f);
        thisPtr = (Object*) GET_REGISTER(vdst & 0x0f);
    }

    /*
     * Combine the object we found with the vtable offset in the
     * method.
     */
    assert(ref < (unsigned int) thisPtr->clazz->vtableCount);
    methodToCall = thisPtr->clazz->vtable[ref]; /* 直接从该变量的实际类型的 vtable
中获取方法 */
    ... ...

    GOTO_invokeMethod(methodCallRange, methodToCall, vsrc1, vdst);
}
GOTO_TARGET_END
```

`invoke-virtual-quick` 效率很明显比 `invoke-virtual` 更高，直接从实际类型的 vtable 中获取调用方法的指针，而省略了 dvmResolveMethod 从变量的引用类型获取该方法在 vtable 索引 ID 的步骤，所以更高效。

现在很容易知道上面代码示例中，方法调用错乱发生的本质原因了。打包前类 A 的 vable 值是 `vtable[0]=a_t2`。打包后类 A 新增了 a_t1 方法，那么类 A 的 vtable 值为 `vtable[0]=a_t1, vtable[1]=a_t2`。但是 obj.a_t2() 这行代码在 odex 中的指令其实是 `invoke-virtual-quick A.vtable[0]`，所以打包前调用的是 a_t2 方法，打包后调用的是 a_t1 方法，导致了方法调用错乱。

3.2.3 终极解决方案

可见，由于多态的影响，QFix 的方案会遇到问题。我们最后的希望就寄托于类似 Tinker 方案的完整 dex 解决方案。

利用 Google 已经开源的 DexMerge 方案，把补丁 dex 和原 dex 合并成一个完整的 dex 似乎是可行的，但仅仅这样还是不够的，多 dex 下如果 DexMerge 抛出 65535 方法数超了的异常，DexMerge 会导致内存风暴，在内存不足的情况下容易更新失败。完整的 dex 合成要求在移动端进行，实现较为复杂。

因此，我们最后自研了一套完善的完整 dex 方案，具体是如何实现的呢？就让我们进入 3.3 节，来揭开这一谜底。

3.3 Dalvik 下完整 dex 方案的新探索

3.3.1 冷启动类加载修复

对于 Android 下的冷启动类加载修复，最早的实现方案是 QQ 空间提出的 dex 插入方案。该方案的主要思想，就是把新 dex 插入到 ClassLoader 索引路径的最前面。这样在加载一个类时，就会优先查找补丁中的类。后来微信的 Tinker 和手 Q 的 QFix 方案都基于该方案做了改进，而这类插入 dex 的方案，都会遇到一个严重的问题，就是如何解决 Dalvik 虚拟机下类的 pre-verify 问题。

如果一个类中直接引用到的所有非系统类都和该类在同一个 dex 中的话，那么这个类就会被打上 `CLASS_ISPREVERIFIED` 标志，具体判定代码可见虚拟机中的 `verifyAndOptimizeClass` 函数。

我们先来看看腾讯的三大热修复方案是如何解决这个问题的：

- QQ 空间的处理方式是在每个类中插入一个来自其他 dex 的 hack.class，由

此让所有类都无法满足 pre-verified 条件。

- Tinker 的方式是合成全量的 dex 文件，这样所有类都在全量 dex 中解决，从而消除类重复而带来的冲突。
- QFix 的方式是获取虚拟机中的某些底层函数，提前解析所有补丁类。以此绕过 Pre-verify 检查。

以上三种方案里，QQ 空间方案会侵入打包流程，并且为了 hack 添加一些臃肿的代码，实现起来很不优雅。而我们一开始采用的 QFix 的方案，需要获取底层虚拟机的函数，不够稳定可靠。并且，和空间方案一样，有个比较大的问题是无法新增 public 函数，具体原因后文会进行详解。

现在看来比较好的方式，就是像 Tinker 那样全量合成完整的新 dex。Tinker 的合成方案是从 dex 的方法和指令维度进行全量合成，虽然可以大大节省空间，但由于对 dex 内容的比较**粒度过细**，实现较为复杂，因此性能消耗比较严重。实际上，dex 的大小占整个 APK 的比例是比较低的，而占空间大的主要还是 APK 中的资源文件。因此，Tinker 方案的时空代价转换的性价比不高。

其实，dex 比较的最佳粒度，应该是在类的维度。它既不像方法和指令维度那样细微，也不像 bsbiff 比较那般粗糙。在类的维度，可以达到时间和空间平衡的最佳效果。

3.3.2 一种新的全量 Dex 方案

一般来说，合成完整 dex，思路就是把原有的 dex 和补丁包里的 dex 重新合并成一个。

然而我们的思路是反过来的。

可以这样考虑，既然补丁中已经有变动的类了，那么只要在原先基线包里的 dex 中，去掉补丁中也有的类。这样，补丁 + 去除了补丁类的基线包，不就等于新 App 中的所有类了吗?

参照 Android 原生 multi-dex 的实现再来看这个方案，会很好理解。multi-dex 是把一个 APK 里用到的所有类拆分到 `classes.dex`、`classes2.dex`、`classes3.dex` 等之中，而每个 dex 都只包含了部分的类定义，但单个 dex 也是可以加载的，因为只要把所有 dex 都加载进去，本 dex 中不存在的类就可以在运行期间在其他的 dex 中找到。

同理，基线包 dex 在去掉了补丁中的类后，原有需要发生变更的类就被消除了，基线包 dex 里就只包含不变的类了。而这些不变的类在要用到补丁中的新类时会自动地找到补丁 dex，补丁包中的新类在需要用到不变的类时也会找到基线包 dex 的类。这样基线包里面不使用补丁类的类仍旧可以按原来的逻辑做 odex，最大程度地保证了 dexopt 的效果。

这么一来，我们不再需要像传统合成思路那样判断类的增加和修改情况，也不需要处理合成时方法数超了的情况，对于 dex 的结构也不用进行破坏性重构。

现在，合成完整 dex 的问题就简化为如何在基线包 dex 里面去掉补丁包中包含的所有类。接下来我们看一下在 dex 中去除指定类的具体实现。

首先，来看 dex 文件中 header 的结构：

```
/*
 * Direct-mApped "header_item" struct.
 */
struct DexHeader {
    uint8_t  magic[8];           /* includes version number */
    uint32_t  checksum;           /* adler32 checksum */
    uint8_t  signature[kSHA1DigestLen]; /* SHA-1 hash */
    uint32_t  fileSize;           /* length of entire file */
    uint32_t  headerSize;         /* offset to start of next section */
    uint32_t  endianTag;
    uint32_t  linkSize;
    uint32_t  linkOff;
    uint32_t  mapOff;
    uint32_t  stringIdsSize;
    uint32_t  stringIdsOff;
    uint32_t  typeIdsSize;
    uint32_t  typeIdsOff;
    uint32_t  protoIdsSize;
```

```
    uint32_t  protoIdsOff;
    uint32_t  fieldIdsSize;
    uint32_t  fieldIdsOff;
    uint32_t  methodIdsSize;
    uint32_t  methodIdsOff;
    uint32_t  classDefsSize;
    uint32_t  classDefsOff;
    uint32_t  dataSize;
    uint32_t  dataOff;
};
```

由 dex header 就可以取得 dex 的各个重要属性，这些属性在文件中的分布如表 3-2[①]所示。

表 3-2　dex 重要属性

名称	格式	说明
header	header_item	标头
string_ids	string_id_item	字符串标识符列表。这些是此文件使用的所有字符串的标识符，用于内部命名（例如类型描述符）或用作代码引用的常量对象。此列表必须使用 UTF-16 代码点值按字符串内容进行排序（不采用语言区域敏感方式），且不得包含任何重复条目
type_ids	type_id_item	类型标识符列表。这些是此文件引用的所有类型（类、数组或原始类型）的标识符（无论文件中是否已定义）。此列表必须按‘string_id’索引进行排序，且不得包含任何重复条目
proto_ids	proto_id_item	方法原型标识符列表。这些是此文件引用的所有原型的标识符。此列表必须按返回类型（按‘type_id’索引排序）主要顺序进行排序，然后按参数列表（按‘type_id’索引排序的各个参数，采用字典排序方法）进行排序。该列表不得包含任何重复条目
field_ids	field_id_item	字段标识符列表。这些是此文件引用的所有字段的标识符（无论文件中是否已定义）。此列表必须进行排序，其中定义类型（按‘type_id’索引排序）是主要顺序，字段名称（按‘string_id’索引排序）是中间顺序，而类型（按‘type_id’索引排序）是次要顺序。该列表不得包含任何重复条目
method_ids	method_id_item	方法标识符列表。这些是此文件引用的所有方法的标识符（无论文件中是否已定义）。此列表必须进行排序，其中定义类型（按‘type_id’索引排序）是主要顺序，方法名称（按‘string_id’索引排序）是中间顺序，而方法原型（按‘proto_id’索引排序）是次要顺序。该列表不得包含任何重复条目

① 此表引用自 https://source.android.com/devices/tech/dalvik/dex-format

（续表）

名称	格式	说明
class_defs	class_def_item	类定义列表。这些类必须进行排序，以便所指定类的超类和已实现的接口比引用类更早出现在该列表中。此外，对于在该列表中多次出现的同名类，其定义是无效的
call_site_ids	call_site_id_item	调用站点标识符列表。这些是此文件引用的所有调用站点的标识符（无论文件中是否已定义）。此列表必须按‘call_site_off’的升序进行排序
method_handles	method_handle_item	方法句柄列表。此文件引用的所有方法句柄的列表（无论文件中是否已定义）。此列表未进行排序，而且可能包含将在逻辑上对应于不同方法句柄实例的重复项
data	ubyte	数据区，包含上面所列表格的所有支持数据。不同的项有不同的对齐要求；如有必要，则在每个项之前插入填充字节，以实现所需的对齐效果
link_data	ubyte	静态链接文件中使用的数据。本文档尚未指定本区段中数据的格式。此区段在未链接文件中为空，而运行时实现可能会在适当的情况下使用这些数据

这里我们打算去除 dex 中的类，因此我们最关心的自然是这里面的 class_defs 属性。

需要注意的是，并不是要把某个类的所有信息都从 dex 移除，因为如果这么做，可能会导致 dex 的各个部分都发生变化，从而需要大量调整 offset，这样就变得就费时费力了。我们要做的，仅仅是使得在解析这个 dex 的时候找不到这个类的定义就可以了。**因此，只需要移除定义的入口，对于类的具体内容不进行删除，这样可以最大限度地减少 `offset` 的修改。**

我们来看虚拟机在 dexopt 的时候是如何找到某个 dex 的所有类定义的。

文件：android-4.4.4_r2/dalvik/vm/analysis/DexPrepare.cpp。

```
/*
 * Verify and/or optimize all classes that were successfully loaded from
 * this DEX file.
 */
static void verifyAndOptimizeClasses(DexFile* pDexFile, bool doVerify,
    bool doOpt)
```

```
{
    u4 count = pDexFile->pHeader->classDefsSize;
    u4 idx;

    for (idx = 0; idx < count; idx++) {
        const DexClassDef* pClassDef;
        const char* classDescriptor;
        ClassObject* clazz;

        pClassDef = dexGetClassDef(pDexFile, idx);
        classDescriptor = dexStringByTypeIdx(pDexFile, pClassDef->classIdx);

        /* all classes are loaded into the bootstrap class loader */
        clazz = dvmLookupClass(classDescriptor, NULL, false);
        if (clazz != NULL) {
            verifyAndOptimizeClass(pDexFile, clazz, pClassDef, doVerify,
                doOpt);

        } else {
            // TODO: log when in verbose mode
            ALOGV("DexOpt: not optimizing unavailable class '%s'",
                classDescriptor);
        }
    }
}
```

正是 `dexGetClassDef` 函数返回了类的定义。

文件：android-4.4.4_r2/dalvik/libdex/DexFile.h。

```
/* return the ClassDef with the specified index */
DEX_INLINE const DexClassDef* dexGetClassDef(const DexFile* pDexFile, u4 idx)
{
    assert(idx < pDexFile->pHeader->classDefsSize);
    return &pDexFile->pClassDefs[idx];
}
```

而这里 pClassDefs 是怎么来的呢?

```
/*
 * Set up the basic raw data pointers of a DexFile. This function isn't
 * meant for general use.
 */
void dexFileSetupBasicPointers(DexFile* pDexFile, const u1* data) {
    DexHeader *pHeader = (DexHeader*) data;

    ... ...
    pDexFile->pClassDefs = (const DexClassDef*) (data + pHeader->classDefsOff);
```

```
    ... ...
}
```

由此可以看出，一个类的所有 DexClassDef，也就是类定义，是从 `pHeader->classDefsOff` 偏移处开始，依次呈线性排列的，一个 dex 里面一共有 `pHeader->classDefsSiz` 个类定义。

由此，我们就可以直接找到 `pHeader->classDefsOff` 偏移处，遍历所有的 `DexClassDef`，如果发现这个 `DexClassDef` 的类名包含在补丁中，就把它移除，实现效果如图 3-2 所示。

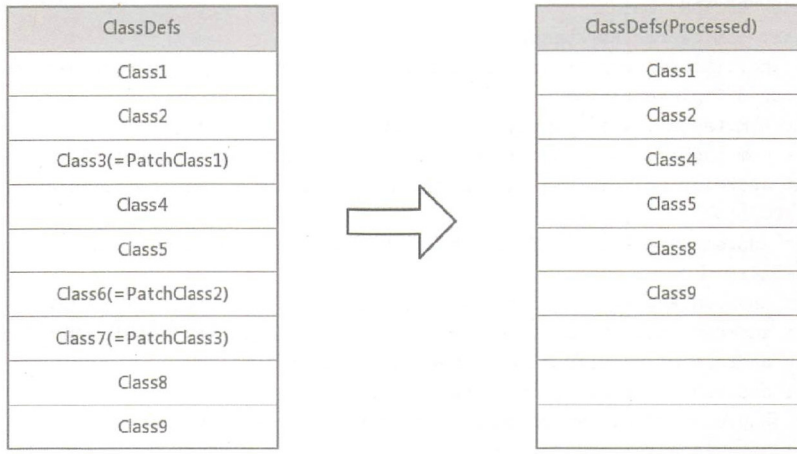


图 3-2　Dex 中 ClassDef 的移除

接下来，只要修改 `pHeader->classDefsSiz`，把 dex 中类的数目改为去除补丁中的类之后的数目即可。

我们只是去除了类的定义，而对于类的方法实体以及其他 dex 信息不做移除，虽然这样会把这个被移除类的无用信息残留在 dex 文件中，但这些信息并不占用太多空间。移除类操作的方式对 dex 的处理速度提升帮助是很大的。

3.3.3 对于 Application 的处理

由此，我们实现了完整的 dex 合成。但仍然有个问题，这个问题所有完整 dex 替换方案都会遇到，那就是对于 Application 的处理。

众所周知，Application 是整个 App 的入口，因此，在进入到替换的完整 dex 之前，一定会通过 Application 的代码，然而，Application 必然是加载在原来的 dex 里面的。只有在补丁加载后使用的类，会在新的完整 dex 里面找到。

因此，在加载补丁后，如果 Application 类使用其他新 dex 里的类，由于不在同一个 dex 里，如果 Application 被打上了 pre-verified 标志，这时就会抛出异常：

```
FATAL EXCEPTION: main
Process: com.taobao.worktest, PID: 2481
java.lang.IllegalAccessError: Class ref in pre-verified class resolved to
unexpected implementation
    at com.taobao.test.MyApplication.test(MyApplication.java:59)
    at com.taobao.test.MyApplication.onCreate(MyApplication.java:39)
    at android.App.Instrumentation.callApplicationOnCreate(Instrumentation.
java:1007)
    at android.App.ActivityThread.handleBindApplication(ActivityThread.
java:4328)
    at android.App.ActivityThread.access$1500(ActivityThread.java:135)
    at android.App.ActivityThread$H.handleMessage(ActivityThread.java:1256)
    at android.os.Handler.dispatchMessage(Handler.java:102)
    at android.os.Looper.loop(Looper.java:136)
    at android.App.ActivityThread.main(ActivityThread.java:5001)
    at java.lang.reflect.Method.invokeNative(Native Method)
    at java.lang.reflect.Method.invoke(Method.java:515)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.
java:785)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:601)
    at dalvik.system.NativeStart.main(Native Method)
```

对此，我们的解决办法很简单，既然被打上了 pre-verified 标志，那么，清除掉它就是了。

类的标志位于 `ClassObject` 的 `accessFlags` 成员中。

```
struct ClassObject : Object {
    /* leave space for instance data; we could access fields directly if we
```

```
        freeze the definition of java/lang/Class */
    u4              instanceData[CLASS_FIELD_SLOTS];

    /* UTF-8 descriptor for the class; from constant pool, or on heap
       if generated ("[C") */
    const char*     descriptor;
    char*           descriptorAlloc;

    /* access flags; low 16 bits are defined by VM spec */
    u4              accessFlags;

    /* VM-unique class serial number, nonzero, set very early */
    u4              serialNumber;

    /* DexFile from which we came; needed to resolve constant pool entries */
    /* (will be NULL for VM-generated, e.g. arrays and primitive classes) */
    DvmDex*         pDvmDex;

    /* state of class initialization */
    ClassStatus     status;

    ... ...
}
```

而 pre-verified 标志的定义是：

```
    CLASS_ISPREVERIFIED         = (1<<16), // class has been pre-verified
```

因此，我们只需要在 JNI 层清除掉它即可：

```
    clazzObj->accessFlags &= ~CLASS_ISPREVERIFIED;
```

这样，在 dvmResolveClass 中找到了新 dex 里的类后，由于 CLASS_ISPRE-VERIFIED 标志被清空，就不会判断所在 dex 是否相同，从而成功避免抛出异常。

```
ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx,
    bool fromUnverifiedConstant) {
    ... ...
        // 条件不满足，不进行 check
        if (!fromUnverifiedConstant &&
            IS_CLASS_FLAG_SET(referrer, CLASS_ISPREVERIFIED))
        {
            ClassObject* resClassCheck = resClass;
            if (dvmIsArrayClass(resClassCheck))
                resClassCheck = resClassCheck->elementClass;
```

```
            // 判断是否在同一 dex 中
            if (referrer->pDvmDex != resClassCheck->pDvmDex &&
                resClassCheck->classLoader != NULL)
            {
                ALOGW("Class resolved by unexpected DEX:"
                     " %s(%p):%p ref [%s] %s(%p):%p",
                    referrer->descriptor, referrer->classLoader,
                    referrer->pDvmDex,
                    resClass->descriptor, resClassCheck->descriptor,
                    resClassCheck->classLoader, resClassCheck->pDvmDex);
                ALOGW("(%s had used a different %s during pre-verification)",
                    referrer->descriptor, resClass->descriptor);
                dvmThrowIllegalAccessError(
                    "Class ref in pre-verified class resolved to unexpected "
                    "implementation");
                return NULL;
            }
        }
    ... ...
```

接下来，我们来对比目前市场上其他完整 dex 方案是怎么做的。

Tinker 的方案是在 `AndroidManifest.xml` 声明中就要求开发者将自己的 Application 直接换成 `TinkerApplication`。而对于真正 App 的 Application，要在初始化 `TinkerApplication` 时作为参数传入。这样 `TinkerApplication` 会接管这个传入的 Application，在生命周期回调时通过反射的方式调用实际 Application 的相关回调逻辑。这么做确实很好地将入口 Application 和用户代码隔离开，不过需要改造原有的 Application，如果对 Application 有更多扩展，接入成本也是比较高的。

Amigo 的方案是在编译过程中，用 Amigo 自定义的 gradle 插件将 App 的 Application 替换成了 Amigo 自己的另一个 Application，并且将原来的 Application 的 name 保存起来，该修复的问题都修复完再调用之前保存的 Application 的 `attach(context)`，然后将它回调到 loadedApk 中，最后调用它的 `onCreate()`，执行原有 Application 中的逻辑。这种方式只是在代码层面开发者无感知，但其实是在编译期间偷偷帮用户做了替换，有点掩耳盗铃的意思，并且这种对系统做反射替换本身也是有一定风险的。

相比之下，我们的 Application 处理方案既没有侵入编译过程，也不需要进行反射替换，所有的兼容操作都在运行期间自动做好。接入过程极其顺滑。

3.3.4 dvmOptResolveClass 问题与对策

然而我们这种清除标志的方案并非一帆风顺，在开发过程中发现，如果这个入口 Application 是**没有 `pre-verified`** 的，反而有更大的问题。

这个问题是，Dalvik 虚拟机如果发现某个类没有 pre-verified，就会在初始化这个类时做 Verify 操作，将会扫描这个类的所有代码，在扫描过程中**对这个类代码里使用到的类**都要进行 `dvmOptResolveClass` 操作。

这个 `dvmOptResolveClass` 正是罪魁祸首，它会在解析的时候对使用到的类进行初始化，而这个逻辑是发生在 Application 类初始化的时候。此时补丁还没进行加载，所以就会提前加载到原始 dex 中的类。接下来当补丁加载完毕后，当这些已经加载的类用到新 dex 中的类，并且又是 pre-verified 时就会报错。

这里最大的问题在于我们无法把补丁加载提前到 dvmOptResolveClass 之前，因为在一个 App 的生命周期里，没有可能到达比入口 Application 初始化更早的时期了。

而这个问题常见于多 dex 情形，当存在多 dex 时，无法保证 Application 用到的类和它处于同个 dex 中。如果只有一个 dex，一般就不会有这个问题。

多 dex 情况下要想解决这个问题，有两种办法：

- 让 Application 用到的所有非系统类都和 Application 位于同一个 dex 里，这就可以保证 pre-verified 标志被打上，避免进入 `dvmOptResolveClass`，而在补丁加载完之后，我们再清除 pre-verified 标志，使得接下来使用其他类也不会报错。
- 把 Application 里面除了热修复框架代码以外的其他代码都剥离开，单独提

出放到一个其他类里面，这样使得 Application 不会直接用到过多非系统类，这样，保证这个单独拿出来的类和 Application 处于同一个 dex 的概率还是比较大的。如果想要更保险，Application 可以采用反射方式访问这个单独的类，这样就彻底把 Application 和其他类隔绝开了。

第一种方法实现较为简单，因为 Android 官方 multi-dex 机制会自动将 Application 用到的类都打包到主 dex 中，所以只要把热修复初始化放在 attachBaseContext 的最前面，一般都没问题。而第二种方法稍加烦琐，是在代码架构层面进行重新设计，不过可以一劳永逸地解决问题。

3.4 入口类与初始化时机的选择

3.4.1 初始化时机

冷启动完整修复方案，本质就是换掉整个原有的 dex 文件。然而“完整替换”只是一种理想化的设想，实际上是无法做到“完整”的。原因是热修复的初始化本身也是一段代码。必须调用到这段代码，热修复操作才能执行完成。因此调用到热修复的类，肯定是使用者自己的类，这个类是无法被热修复影响到的，并且它只能存在于原始安装包的 classes.dex 中。如果要使热修复类之前使用的其他类最少，只能放在 Application 类入口中。

那么，放在 Activity 类里面是不是也可以呢？当然，如果你的 App 里面没有 Application，放到 Activity 里面似乎没有太大问题，并且简单测试好像也能正常工作。

但是，如果你的 AndroidManifest 中注册了 ContentProvider，事情就没那么顺利了。ContentProvider 的 onCreate 方法的调用先于 Activity 的 onCreate 方法。这就使得我们可能还没完成热修复替换，就先执行到了 ContentProvider 中的业务逻辑代码，导致某些类被提前引入。提前引入其他类的危害我们在之前的章节已

说明，这不仅会导致这些类无法修复，更可能引起 pre-verify 异常。因此，只有把初始化放在 Application 类中，才能保证不会错误地提早引入类。

如果放在 Application 中，又有两种选择：放在 onCreate 中或者放在 attachBaseContext 中。

放在 attachBaseContext 中自然是没问题的，因为它是 Application 中最早被执行的代码，但需要注意的是，在 attachBaseContext 里面有很多限制，此时 App 申请的权限还没授予完成，所以会遇到无法访问网络之类的问题。因此，在 attachBaseContext 里面可以执行初始化，但不可以进行网络请求下载新补丁。

那放在 Application 的 onCreate 里面可以吗？简单测试似乎没有什么问题。然而，它和之前的 Activity 的 onCreate 方法一样，执行时间会晚于 ContentProvider 的 onCreate 方法。

这虽然不太符合个人直觉，但真实的启动顺序确实是按照如下顺序进行的：Application.attachBaseContext → ContentProvider.onCreate → Application.onCreate → Activity.onCreate。感兴趣的读者可以自行研读系统源码来验证这一执行顺序。

当然，如果你的 AndroidManifest 里面没有注册过 ContentProvider，并且能够保证引用的第三方库的 AndroidManifest 里面也没有注册，放在 onCreate 里面就没有什么问题。不过保险起见，为了避免以后某天项目在无意中引入，还是放在 attachBaseContext 里面最好。

3.4.2 防不胜防的细节错误

在进行初始化的时候，经常容易错误地提早引入其他类。

下面这段代码是 Sophix 热修复初始化的代码，`SophixManager` 需要在设置各种属性后调用 initialize 方法进行初始化，就以这段代码为例：

```
public class SampleApplication extends Application {
    LocalStorageUtil localStorageUtil = new LocalStorageUtil();

    protected void attachBaseContext(Context base) {
        CrashReport.initCrashReport(this);
        SophixWrApper.init(this);
        MultiDex.install(this);
        localStorageUtil.init(this);
    }

    public void onCreate() {
        super.onCreate();
        SophixWrApper.query();
    }

    public LocalStorageUtil getLocalStorageUtil() {
        return localStorageUtil;
    }

    static private class SophixWrApper {
        static void init(Application context) {
            final SophixManager instance = SophixManager.getInstance();
            instance.setContext(context)
                    .setAppVersion(BuildConfig.VERSION_NAME)
                    .setPatchLoadStatusStub(new PatchLoadStatusListener() {
                        @Override
                        public void onLoad(final int mode,
                                           final int code,
                                           final String info,
                                           final int handlePatchVersion) {
                            if (code == PatchStatus.CODE_LOAD_SUCCESS) {
                                MyLogger.i("", "sophix load patch success!");
                            } else if (code == PatchStatus.CODE_LOAD_
                                RELAUNCH) {
                                MyLogger.i("", "sophix preload patch
                                success.");
                            }
                        }
                    });
            instance.initialize();
        }

        static void query() {
            SophixManager.getInstance().queryAndLoadNewPatch();
        }
    }
}
```

这段简单的代码里面，包含了许多开发者都会出现的错误。在继续阅读下文之前，读者可以先浏览可能会出现的几个问题：

1. CrashReport.initCrashReport(this) 在 Sophix 热修复初始化之前提早引入了，必然是不行的。

2. 虽然初始化确实是在 attachBaseContext 里面，但是包装了一个 SophixWrApper 类，这会导致初始化之前提前引入类。因此 Sophix 的初始化不可以包装在其他类中。

3. 在 setAppVersion 的时候使用了 BuildConfig 类，这个 BuildConfig 类是 Android 编译期间动态生成的，也属于非系统类，如果在这里使用就会有提前引入的问题。这里建议用 PackageManager 来获取版本号。

4. 在回调类中使用了 MyLogger，在回调状态的时候引用很可能热修复还未初始化完毕，因此这里需要换为系统类 android.utils.log。

5. LocalStorageUtil 直接在声明处赋值了它的实例，这个赋值其实是隐式发生在对象构造函数中的，这个时候甚至是早于 attachBaseContext 的，因此也是不行的，需要在初始化之后才能进行赋值。

6. MultiDex.install(this) 调用放在了热修复初始化后，这样做虽然没有引入类的问题，但是可能会导致后面热修复框架初始化的时候找不到其他不在主 dex 中的热修复框架内部类，因此需要把它提前到热修复初始化之前。而提早引入 MultiDex 类不会带来问题，因为在热修复初始化之后，再也没有调用到这个 MultiDex 类的地方。

7. 最后，经常会有人遗漏了 super.attachBaseContext(base)，如果缺少它，后面都无法正常运行。

现在来看一下修改后的代码：

```
public class SampleApplication extends Application {
    LocalStorageUtil localStorageUtil;
```

```
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    MultiDex.install(this);
    initSophix(this);
    CrashReport.initCrashReport(this);
    initlocalStorageUtil();
}

public void onCreate() {
    super.onCreate();
    SophixManager.getInstance().queryAndLoadNewPatch();
}

private void initlocalStorageUtil() {
    localStorageUtil = new LocalStorageUtil();
    localStorageUtil.init(this);
}

public LocalStorageUtil getLocalStorageUtil() {
    return localStorageUtil;
}

private void initSophix(Application context) {
    String AppVersion = "0";
    try {
        AppVersion = this.getPackageManager()
                .getPackageInfo(this.getPackageName(), 0)
                .versionName;
    } catch (Exception ignored) {
    }
    final SophixManager instance = SophixManager.getInstance();
    instance.setContext(context)
            .setAppVersion(AppVersion)
            .setAesKey(null)
            .setEnableDebug(false)
            .setEnableFullLog()
            .setPatchLoadStatusStub(new PatchLoadStatusListener() {
                @Override
                public void onLoad(final int mode,
                                   final int code,
                                   final String info,
                                   final int handlePatchVersion) {
                    if (code == PatchStatus.CODE_LOAD_SUCCESS) {
                        Log.i("", "sophix load patch success!");
                    } else if (code == PatchStatus.CODE_LOAD_RELAUNCH) {
                        Log.i("", "sophix preload patch success.");
                    }
                }
```

```
                });
            instance.initialize();
        }
    }
```

这样就万无一失了。这里初始化放到独立的 initSophix 是没关系的，因为是入口 Application 自己的方法，所以不会新引入任何其他类。

3.4.3 入口类带来的修复限制

在入口类中初始化，即使保证了初始化完全正确，仍然还有一个限制，那就是如果修改了入口 Application 中直接使用的类的结构，很有可能会引起错位异常。

就以上一节代码里面的 initlocalStorageUtil 为例。

```
    private void initlocalStorageUtil() {
        localStorageUtil = new LocalStorageUtil();
        localStorageUtil.init(this);
    }
```

它的 dex 字节码是这样的：

```
0x0000: new-instance v0, com.taobao.worktest.LocalStorageUtil // type@2417
0x0002: invoke-direct {v0}, void com.taobao.worktest.
LocalStorageUtil.<init>() // method@18715
0x0005: iput-object v0, v1, Lcom/taobao/worktest/LocalStorageUtil; com.
taobao.worktest.SampleApplication.localStorageUtil // field@9182
0x0007: iget-object v0, v1, Lcom/taobao/worktest/LocalStorageUtil; com.
taobao.worktest.SampleApplication.localStorageUtil // field@9182
0x0009: invoke-virtual {v0, v1}, void com.taobao.worktest.LocalStorageUtil.
init(com.taobao.worktest.SampleApplication) // method@18717
0x000c: return-void
```

而最终执行的时候，大多数时候是以 oat 机器码的方式执行的。这里当引用 localStorageUtil 的 init 方法时，如果是根据 method id 直接从 dex 相关数据中获取 ArtMethod 结构然后执行，是没问题的。

但是，如果 oat 文件做过比较大的优化，变为直接从 localStorageUtil 对象的所

属类 LocalStorageUtil 里面去查找 init 方法，这个时候可能就有问题。

如果获取某个类的某个方法，是根据这个方法在类里面的方法索引来取得的，这个索引是这个方法在类里面的序号，那么，如果在这个类里面新增或者减少方法，就会导致这个类中的方法索引与原有的不一致。而在 Application 入口类中，仍然是处于安装包的 oat 文件里，是原有的索引，所以如果用这个索引去类中获取方法，将可能不再是原来的方法，从而引发崩溃。同理，对于类里面的字段，也存在索引，因此也有类似的问题。

不过，触发这个问题需要使得 Application 中使用的类的方法索引发生变化。如果对这些使用的类，不增加或者减少方法数或者字段的话，就没关系。即使发生了增减的情况，如果在 Application 里面直接使用到的这些方法或者字段索引没有受到影响，那也是没问题的。例如插入的方法正好是在所用方法之后，那就影响不到这个方法的索引。保险起见，还是需要注意这类情况，并尽量避免。

正是基于这种限制，我们后面引出了一种新的更加稳健的初始化方式，保证初始化在单独的入口类中进行，后面再用反射的方式替换为原有的 Application。采用新的初始化方式，之前的 SampleApplication 可以改造成这样：

```
public class SampleApplication extends Application {
    LocalStorageUtil localStorageUtil;

    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        CrashReport.initCrashReport(this);
        initlocalStorageUtil();
    }

    public void onCreate() {
        super.onCreate();
        SophixManager.getInstance().queryAndLoadNewPatch();
    }

    private void initlocalStorageUtil() {
        localStorageUtil = new LocalStorageUtil();
        localStorageUtil.init(this);
    }
```

```
    public LocalStorageUtil getLocalStorageUtil() {
        return localStorageUtil;
    }
}
```

这里我们把 Sophix 初始化相关逻辑移除，而把初始化放到了一个单独的 SophixStubApplication 类中，这个类会作为 AndroidManifest 的入口替换掉原有的 SampleApplication 类。

```
public class SophixStubApplication extends SophixApplication {
    private final String TAG = "SophixStubApplication";

    /* 此处 SophixEntry 应指定真正的 Application，并且保证 RealApplicationStub 类名不
       被混淆 */
    @Keep
    @SophixEntry(SampleApplication.class)
    static class RealApplicationStub {}
    @Override

    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        MultiDex.install(this);
        initSophix();
    }

    private void initSophix(Application context) {
        String AppVersion = "0";
        try {
            AppVersion = this.getPackageManager()
                    .getPackageInfo(this.getPackageName(), 0)
                    .versionName;
        } catch (Exception ignored) {
        }
        final SophixManager instance = SophixManager.getInstance();
        instance.setContext(context)
                .setAppVersion(AppVersion)
                .setAesKey(null)
                .setEnableDebug(false)
                .setEnableFullLog()
                .setPatchLoadStatusStub(new PatchLoadStatusListener() {
                    @Override
                    public void onLoad(final int mode,
                                       final int code,
                                       final String info,
```

```
                                        final int handlePatchVersion) {
                        if (code == PatchStatus.CODE_LOAD_SUCCESS) {
                            Log.i("", "sophix load patch success!");
                        } else if (code == PatchStatus.CODE_LOAD_RELAUNCH) {
                            Log.i("", "sophix preload patch success.");
                        }
                    }
                });
        instance.initialize();
    }
}
```

这便是我们的单独入口类。它不包含任何的业务逻辑，只有负责 Sophix 初始化与替换 Application 的操作。替换逻辑在它的父类 SophixApplication 中实现。这里的 SophixEntry 设为原有的 SampleApplication，以便于我们知道需要换的 Application 是什么。而 MultiDex 也需要放到这里来初始化，因为可能如果有多 dex 的情况，Sophix 里的其他类很可能不完全在主 dex 里面。

开发者使用这种方式进行初始化的时候，只需要复制这个 SophixStubApplication 类到自己的项目中，然后把 AndroidManifest 里面的 Application 指定为它，再设置 SophixEntry 为 SampleApplication 就可以了。

Sophix 在运行的时候，会先执行初始化逻辑，当初始化完成后，通过反射得到 SophixStubApplication 的静态内部类 RealApplicationStub，最终通过它的类注解 SophixEntry 得到真正的 Application 即 SampleApplication。然后调用 SampleApplication 的生命周期函数 attachBaseContext、onCreate 等，再进行替换。后续，所有使用 Application 的地方都能够找到这个换回来的 SampleApplication。

其他热修复方案也有很多采用的是这种替换 Application 的方式。然而，它们主要的实现方式都是在编译期间，通过 gradle 插件偷偷地把 AndroidManifest 中的入口 Application 换掉。相比之下，我们这种方式，似乎更加优雅一些，对开发者做到了“无感知”。

然而我们其实不推崇这种掩耳盗铃的行为，我们是直接提示开发者自己替换这

个 Application 入口，把细节更多地暴露在阳光下。这样做还有一个好处，即可以避免修改打包插件，因为根本不存在打包插件，这也是无侵入思想的体现。

在此，需要重申一下不侵入编译流程带来的巨大好处。首先，开发者在编译期间可以直接使用 Android 原生插件或自定义插件，不限于任何 IDE 和打包工具。并且，当 Android 原生编译链升级时，不必对新版本 gradle 进行适配，也不会受到其他 JVM 平台新语言的加入（如 Kotlin）的影响。由于直接作用于 APK 产物，我们的打包工具永远可以保持稳定可用，无缝兼容。

3.5 本章小结

本章主要讲解了代码冷启动类加载修复的实现方式，冷启动虽然没有热替换的高即时性，但其稳定性好、修复范围广，更适用于一般的线上生产环境。冷启动修复也有一些需要注意的事项，主要在于避免提前引入非系统 API 类。Sophix 采用了冷热两种方案相结合的方式，实现了按需选择、完全兼顾的效果。

第 4 章
资源热修复技术

资源修复的技术细节与思考。

4.1 普遍的实现方式

Android 资源热修复，是指在 App 不重新安装的情况下，利用下发的补丁包直接更新 App 中的**资源**。

目前市面上的很多资源热修复方案基本上都参考了 Instant Run 的实现。

首先，我们简单来看一下 Instant Run 是怎么做到资源热修复的。

Instant Run 资源热修复的核心代码就是 monkeyPatchExistingResources 方法。

文件：com/android/tools/fd/runtime/MonkeyPatcher.java。

```
public static void monkeyPatchExistingResources(@Nullable Context context,
                                                 @Nullable String
externalResourceFile,
                                                 @Nullable
Collection<Activity> activities) {

    if (externalResourceFile == null) {
        return;
    }

    try {
        /* %% Part 1. 创建一个新的 AssetManager，并通过反射调用 addAssetPath 添加 /
           sdcard 上的新资源包 */
        // 这样就构造出了一个带新资源的 AssetManager
        /* Create a new AssetManager instance and point it to the resources
           installed under*/
        // /sdcard
        AssetManager newAssetManager = AssetManager.class.getConstructor()
                                                  .newInstance();
        Method mAddAssetPath = AssetManager.class
```

```
                                .getDeclaredMethod("addAssetPath", String.
                                class);
        mAddAssetPath.setAccessible(true);
        if (((Integer) mAddAssetPath.invoke(newAssetManager,
            externalResourceFile)) == 0) {
            throw new IllegalStateException("Could not create new
            AssetManager");
        }

        /* Kitkat needs this method call, Lollipop doesn't. However, it
           doesn't seem to cause any harm*/
        // in L, so we do it unconditionally.
        Method mEnsureStringBlocks = AssetManager.class
                                        .getDeclaredMethod
                                        ("ensureStringBlocks");
        mEnsureStringBlocks.setAccessible(true);
        mEnsureStringBlocks.invoke(newAssetManager);

        /* %% Part 2. 反射得到 Activity 中 AssetManager 的引用处，全部换成刚才新构建的
           newAssetManager*/
        if (activities != null) {
            for (Activity activity : activities) {
                Resources resources = activity.getResources();

                try {
                    Field mAssets = Resources.class
                                        .getDeclaredField("mAssets");
                    mAssets.setAccessible(true);
                    mAssets.set(resources, newAssetManager);
                } catch (Throwable ignore) {
                    Field mResourcesImpl = Resources.class
                                                .getDeclaredField
                                                ("mResourcesImpl");
                    mResourcesImpl.setAccessible(true);
                    Object resourceImpl = mResourcesImpl.get(resources);
                    Field implAssets = resourceImpl.getClass()
                                                .getDeclaredField("mAssets");
                    implAssets.setAccessible(true);
                    implAssets.set(resourceImpl, newAssetManager);
                }
                    ... ...

                pruneResourceCaches(resources);
            }
        }

        /* %% Part 3. 得到 Resources 的弱引用集合，把它们的 AssetManager 成员替换成
           newAssetManager*/
        // Iterate over all known Resources objects
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
Collection<WeakReference<Resources>> references;
if (SDK_INT >= KITKAT) {
    // Find the singleton instance of ResourcesManager
    Class<?> resourcesManagerClass = Class.forName("android.App
                                        .ResourcesManager");
    Method mGetInstance = resourcesManagerClass
                              .getDeclaredMethod("getInstance");
    mGetInstance.setAccessible(true);
    Object resourcesManager = mGetInstance.invoke(null);
    try {
        Field fMActiveResources = resourcesManagerClass
                                  .getDeclaredField
                                  ("mActiveResources");
        fMActiveResources.setAccessible(true);
        @SuppressWarnings("unchecked")
        ArrayMap<?, WeakReference<Resources>> arrayMap =
                (ArrayMap<?, WeakReference<Resources>>)
        fMActiveResources.get(resourcesManager);
        references = arrayMap.values();
    } catch (NoSuchFieldException ignore) {
        Field mResourceReferences = resourcesManagerClass
                                    .getDeclaredField
                                    ("mResourceReferences");
        mResourceReferences.setAccessible(true);
        //noinspection unchecked
        references = (Collection<WeakReference<Resources>>)
        mResourceReferences.get(resourcesManager);
    }
} else {
    Class<?> activityThread = Class.forName("android.App
                                  .ActivityThread");
    Field fMActiveResources = activityThread
                                  .getDeclaredField
                                  ("mActiveResources");
    fMActiveResources.setAccessible(true);
    Object thread = getActivityThread(context, activityThread);
    @SuppressWarnings("unchecked")
    HashMap<?, WeakReference<Resources>> map =
            (HashMap<?, WeakReference<Resources>>) fMActiveResources
                .get(thread);
    references = map.values();
}
for (WeakReference<Resources> wr : references) {
    Resources resources = wr.get();
    if (resources != null) {
        /* Set the AssetManager of the Resources instance to our
           brand new one*/
        try {
            Field mAssets = Resources.class
```

```
                                    .getDeclaredField("mAssets");
                mAssets.setAccessible(true);
                mAssets.set(resources, newAssetManager);
            } catch (Throwable ignore) {
                Field mResourcesImpl = Resources.class
                                            .getDeclaredField
                                            ("mResourcesImpl");
                mResourcesImpl.setAccessible(true);
                Object resourceImpl = mResourcesImpl.get(resources);
                Field implAssets = resourceImpl.getClass()
                                        .getDeclaredField("mAssets");
                implAssets.setAccessible(true);
                implAssets.set(resourceImpl, newAssetManager);
            }

            resources.updateConfiguration(resources.getConfiguration(),
                                          resources.getDisplayMetrics());
        }
    }
} catch (Throwable e) {
    throw new IllegalStateException(e);
}}
```

简要说来，Instant Run 中的资源热修复分为两步：

1. 构造一个新的 AssetManager，并通过反射调用 addAssetPath，把这个完整的新资源包加入到 AssetManager 中。这样就得到了一个含有所有新资源的 AssetManager。
2. 找到所有之前引用到原有 AssetManager 的地方，通过反射，把引用处替换为 AssetManager。

其实仔细看可以发现，大量代码都是在处理兼容性问题，并找到**所有** AssetManager 的引用处。真正的实现逻辑其实很简单。

这其中的重点，自然是 addAssetPath 这个函数。现在我们来看一下它的底层实现逻辑。

以 Android 6.0 为例，addAssetPath 最终调用到了 native 方法。

文件：frameworks/base/core/java/android/content/res/AssetManager.java。

仅供非商业用途或交流学习使用

```
/**
 * Add an additional set of assets to the asset manager.  This can be
 * either a directory or ZIP file.  Not for use by Applications.
   Returns
 * the cookie of the added asset, or 0 on failure.
 * {@hide}
 */
public final int addAssetPath(String path) {
   synchronized (this) {
       int res = addAssetPathNative(path);
       makeStringBlocks(mStringBlocks);
       return res;
   }
}

... ...

private native final int addAssetPathNative(String path);
```

Java 层的 AssetManager 只是个包装，真正关于资源处理的所有逻辑，其实都位于 native 层由 C++ 实现的 AssetManager。

执行 addAssetPath 就是解析这个格式，然后构造出底层数据结构的过程。整个解析资源的调用链是：

- public final int addAssetPath(String path)
- android_content_AssetManager_addAssetPath
- AssetManager::addAssetPath
- AssetManager::AppendPathToResTable
- ResTable::add
- ResTable::addInternal
- ResTable::parsePackage

解析的细节比较烦琐，就不细细说明了，有兴趣的读者可以一层层深究下去。

大致过程就是，通过传入的资源包路径，先得到其中的 resources.arsc，然后解析它的格式，存放在底层的 AssetManager 的 mResources 成员中。

仅供非商业用途或交流学习使用

```
@frameworks/base/include/androidfw/AssetManager.h
class AssetManager : public AAssetManager {

    ... ...

    mutable ResTable* mResources;

    ... ...
```

AssetManager 的 mResources 成员是一个 ResTable 结构体：

```
class ResTable
{
    mutable Mutex               mLock;
    // 互斥锁，用于多进程间互斥操作

    status_t                    mError;

    ResTable_config             mParams;

    // Array of all resource tables.
    Vector<Header*>             mHeaders;
    /* 表示所有 resources.arsc 原始数据，这就等同于所有通过 addAssetPath 加载进来的路径
       的资源 ID 信息 */

    // Array of packages in all resource tables.
    Vector<PackageGroup*>       mPackageGroups;
    // 资源包的实体，包含所有加载进来的 package id 所对应的资源

    // MApping from resource package IDs to indices into the internal
    // package array.
    uint8_t                     mPackageMap[256];
    /* 索引表，表示 0~255 的 package id，每个元组分别存放 该 ID 所属 PackageGroup 在
       mPackageGroups 中的 index*/

    uint8_t                     mNextPackageId;
};
```

一个 Android 进程只包含一个 ResTable，ResTable 的成员变量 mPackageGroups 就是所有解析过的资源包的集合。任何一个资源包中都含有 resources.arsc，它记录了所有资源的 ID 分配情况，以及资源中的所有字符串。这些信息是以二进制数的方式存储的。底层的 AssetManager 做的事就是解析这个文件，然后把相关信息存储到 mPackageGroups 里面。

仅供非商业用途或交流学习使用

4.2 资源文件的格式

整个 resources.arsc 文件，实际上是由一个个 ResChunk（以下简称 chunk）拼接起来的。从文件头开始，每个 chunk 的头部都是一个 ResChunk_header 结构，它指示了这个 chunk 的大小和数据类型。

```
/**
 * Header that Appears at the front of every data chunk in a resource.
 */
struct ResChunk_header
{
    // Type identifier for this chunk.  The meaning of this value depends
    // on the containing chunk.
    uint16_t type;

    // Size of the chunk header (in bytes).  Adding this value to
    // the address of the chunk allows you to find its associated data
    // (if any).
    uint16_t headerSize;

    // Total size of this chunk (in bytes).  This is the chunkSize plus
    // the size of any data associated with the chunk.  Adding this value
    // to the chunk allows you to completely skip its contents (including
    // any child chunks).  If this value is the same as chunkSize, there is
    // no data associated with the chunk.
    uint32_t size;
};
```

通过 ResChunk_header 中的 type 成员，可以知道这个 chunk 是什么类型，从而就可以知道应该如何解析这个 chunk。

解析完一个 chunk 后，从这个 chunk + size 的位置开始，就可以得到下一个 chunk 起始位置，这样就可以依次读取完整个文件的数据内容。

一般来说，一个 resources.arsc 里面包含若干个 package，不过默认情况下，由打包工具 AAPT 打出来的包只有一个 package。这个 package 里包含了 App 中的所有资源信息。

资源信息主要是指每个资源的名称以及它对应的编号。我们知道，Android 中的每个资源，都有它唯一的编号。

仅供非商业用途或交流学习使用

编号是一个 32 位数字，用十六进制来表示就是 0xPPTTEEEE。PP 为 package id，TT 为 type id，EEEE 为 entry id。

它们代表什么？在 resources.arsc 里是以怎样的方式记录的呢？

- 对于 package id，每个 package 对应的是类型为 RES_TABLE_PACKAGE_TYPE 的 ResTable_package 结构体，ResTable_package 结构体的 ID 成员变量就表示它的 package id。
- 对于 type id，每个 type 对应的是类型为 RES_TABLE_TYPE_SPEC_TYPE 的 ResTable_typeSpec 结构体。它的 ID 成员变量就是 type id。但是，该 type id 具体对应什么类型，是需要到 package chunk 里的 Type String Pool 中去解析得到的。比如 Type String Pool 中依次有 attr、drawable、mipmap、layout 字符串，就表示 attr 类型的 type id 为 1，drawable 类型的 type id 为 2，mipmap 类型的 type id 为 3，layout 类型的 type id 为 4。所以，每个 type id 对应了 Type String Pool 里的字符顺序所指定的类型。
- 对于 entry id，每个 entry 表示一个资源项，资源项是按照排列的先后顺序自动被标记编号的。也就是说，一个 type 里按位置出现的第一个资源项，其 entry id 为 0x0000，第二个为 0x0001，依此类推。因此我们是无法直接指定 entry id 的，只能够根据排布顺序决定。资源项之间是紧密排布的，没有空隙，但是可以指定资源项为 ResTable_type::NO_ENTRY 来填入一个空资源。

举个例子，我们随便找个带资源的 APK，用 AAPT 解析一下，看到其中的一行是：

```
$ aapt d resources App-debug.apk

... ...
      spec resource 0x7f040019 com.taobao.patch.demo:layout/activity_main:
flags=0x00000000
... ...
```


仅供非商业用途或交流学习使用

这就表示，activity_main.xml 这个资源的编号是 0x7f040019。它的 package id 是 0x7f，资源类型的 ID 为 0x04，Type String Pool 里的第四个字符串正是 layout 类型，而 0x04 类型的第 0x0019 个资源项就是 activity_main 这个资源。

4.3 运行时资源的解析

默认由 Android SDK 编出来的 APK，是由 AAPT 工具进行打包的，其资源包的 package id 就是 0x7f。

系统的资源包，也就是 framework-res.jar，package id 为 0x01。

在走到 App 的第一行代码之前，系统就已经帮我们构造好一个已经添加了安装包资源的 AssetManager 了。

```
@frameworks/base/core/java/android/App/ResourcesManager.java

    Resources getTopLevelResources(String resDir, String[] splitResDirs,
            String[] overlayDirs, String[] libDirs, int displayId,
            Configuration overrideConfiguration, CompatibilityInfo
compatInfo) {
        ... ...

        AssetManager assets = new AssetManager();
        // resDir 就是安装包 APK
        if (resDir != null) {
            if (assets.addAssetPath(resDir) == 0) {
                return null;
            }
        }

        ... ...
```

因此，这个 AssetManager 里就已经包含了系统资源包及 App 的安装包，就是 package id 为 0x01 的 framework-res.jar 中的资源和 package id 为 0x7f 的 App 安装包资源。

如果此时直接在原有 AssetManager 上继续 addAssetPath 的完整补丁包的

仅供非商业用途或交流学习使用

话，由于补丁包里面的 package id 也是 0x7f，就会使得同一个 package id 的包被加载两次。这会有怎样的问题呢?

在 Android L 之后，这是没问题的，它会默默地把后来的包添加到之前的包的同一个 PackageGroup 下面。

而在解析的时候，会与之前的包比较同一个 type id 所对应的类型，如果该类型下的资源项数目和之前添加过的不一致，会打出一条 warning log，但是仍旧加入到该类型的 TypeList 中。

```
status_t ResTable::parsePackage(const ResTable_package* const pkg,
                                const Header* const header)
    ... ...
                TypeList& typeList = group->types.editItemAt(typeIndex);
                if (!typeList.isEmpty()) {
                    const Type* existingType = typeList[0];
                    if (existingType->entryCount != newEntryCount &&
                        idmapIndex < 0) {
                        ALOGW("ResTable_typeSpec entry count inconsistent:
                        given %d, previously %d",
                                (int) newEntryCount, (int) existingType-
                                >entryCount);
                        /* We should normally abort here, but some legacy
                           Apps declare*/
                        /* resources in the 'android' package (old bug in
                           AAPT).*/
                    }
                }

                Type* t = new Type(header, package, newEntryCount);
                t->typeSpec = typeSpec;
                t->typeSpecFlags = (const uint32_t*)(
                        ((const uint8_t*)typeSpec) + dtohs(typeSpec->header.
                                                    headerSize));
                if (idmapIndex >= 0) {
                    t->idmapEntries = idmapEntries[idmapIndex];
                }
                typeList.add(t);
    ... ...
```

但是在获取这个资源的时候呢?

```
status_t ResTable::getEntry(
```


仅供非商业用途或交流学习使用

```
        const PackageGroup* packageGroup, int typeIndex, int entryIndex,
        const ResTable_config* config,
        Entry* outEntry) const
{
    const TypeList& typeList = packageGroup->types[typeIndex];
    ... ...

    // %% 从第一个 type 开始遍历，也就是说会先取得安装包的资源，然后才是补丁包的
    // Iterate over the Types of each package.
    const size_t typeCount = typeList.size();
    for (size_t i = 0; i < typeCount; i++) {
        const Type* const typeSpec = typeList[i];

        int realEntryIndex = entryIndex;
        int realTypeIndex = typeIndex;
        bool currentTypeIsOverlay = false;

        if (static_cast<size_t>(realEntryIndex) >= typeSpec->entryCount) {
            ALOGW("For resource 0x%08x, entry index(%d) is beyond type
            entryCount(%d)",
                    Res_MAKEID(packageGroup->id - 1, typeIndex, entryIndex),
                    entryIndex, static_cast<int>(typeSpec->entryCount));
            // We should normally abort here, but some legacy Apps declare
            // resources in the 'android' package (old bug in AAPT).
            continue;
        }

        const size_t numConfigs = typeSpec->configs.size();
        for (size_t c = 0; c < numConfigs; c++) {
            ... ...

            if (bestType != NULL) {
                /* Check if this one is less specific than the last found.
                   If so,*/
                /* we will skip it.  We check starting with things we most
                   care*/
                // about to those we least care about.
                if (!thisConfig.isBetterThan(bestConfig, config)) {
                    if (!currentTypeIsOverlay || thisConfig.
                            compare(bestConfig) != 0) {
                        continue;
                    }
                }
            }

            bestType = thisType;
            bestOffset = thisOffset;
            bestConfig = thisConfig;
            bestPackage = typeSpec->package;
```

```
            actualTypeIndex = realTypeIndex;

            // If no config was specified, any type will do, so skip
            if (config == NULL) {
                break;
            }
        }
    }
}
```

在获取某个 Type 的资源时，会从前往后遍历，也就是说先得到原有安装包里的资源，除非后面的资源的 config 比前面的更详细才会发生覆盖。而对于同一个 config 而言，补丁中的资源就永远无法生效了。所以在 Android L 以上的版本，在原有 AssetManager 上加入补丁包，是没有任何作用的，补丁中的资源无法生效。

而在 Android 4.4 及以下版本，addAssetPath 只是把补丁包的路径添加到了 mAssetPath 中，而真正解析的资源包的逻辑是在 App 第一次执行 AssetManager::getResTable 的时候。

文件：android-4.4.4_r2/frameworks/base/libs/androidfw/AssetManager.cpp。

```
const ResTable* AssetManager::getResTable(bool required) const
{
    // %% mResources 已存在，直接返回，不再往下走。
    ResTable* rt = mResources;
    if (rt) {
        return rt;
    }

    // Iterate through all asset packages, collecting resources from each.

    AutoMutex _l(mLock);

    if (mResources != NULL) {
        return mResources;
    }

    if (required) {
        LOG_FATAL_IF(mAssetPaths.size() == 0, "No assets added to
        AssetManager");
    }

    if (mCacheMode != CACHE_OFF && !mCacheValid)
```

```
        const_cast<AssetManager*>(this)->loadFileNameCacheLocked();

    const size_t N = mAssetPaths.size();
    for (size_t i=0; i<N; i++) {
        // ... %% 真正解析 package 的地方 ...
    }

    if (required && !rt) ALOGW("Unable to find resources file resources.
        arsc");
    if (!rt) {
        mResources = rt = new ResTable();
    }
    return rt;
}
```

而在执行到加载补丁代码的时候，getResTable 已经执行过了无数次。这是因为就算我们之前没做过任何资源相关操作，Android framework 里的代码也会多次调用到那里。所以，以后即使是 addAssetPath，也只是添加到了 mAssetPath，并不会发生解析。因而补丁包里面的资源是完全不生效的！

所以，像 Instant Run 这种方案，一定需要一个全新的 AssetManager 时，再加入完整的新资源包，替换掉原有的 AssetManager。

4.4 另辟蹊径的资源修复方案

一个好的资源热修复方案又是怎样的呢?

首先，补丁包要足够小，像直接下发完整的补丁包肯定是不行的，很占用空间。

而像有些方案，是先进行 bsdiff，对资源包做差量处理，然后下发差量包，在运行时合成完整包再加载。这样确实减小了包的体积，却在运行时多了合成的操作，耗费了运行时间和内存。合成后的包也是完整的包，仍旧会占用磁盘空间。

而如果不采用类似 Instant Run 的方案，市面上许多实现方案是自己修改 AAPT，在打包时将补丁包资源进行重新编号。这样就会涉及修改 Android SDK 工

具包，既不利于集成也无法很好地对将来的 AAPT 版本进行升级。

针对以上几个问题，一个好的资源热修复方案，既要保证补丁包足够小，不在运行时占用很多资源，又要保证不侵入打包流程。因此我们提出了一个目前市面上未曾实现的方案。

简单来说，我们构造了一个 package id 为 0x66 的资源包，这个包里只包含改变了的资源项，直接在原有 AssetManager 中 addAssetPath 这个包即可。

真的这么简单?

没错！由于补丁包的 package id 为 0x66，不与目前已经加载的 0x7f 冲突，因此直接加入到已有的 AssetManager 中就可以直接使用了。补丁包里面的资源，只包含原有包里面**没有**而新的包里面**有**的新增资源，以及原有内容发生了改变的资源。

而资源的改变包含增加、减少、修改这三种情况，我们分别是如何处理的呢?

- 对于新增资源，直接加入补丁包，然后新代码里直接引用就可以了，没什么好说的。
- 对于减少资源，我们只要不使用它就行了，因此不用考虑这种情况，它也不影响补丁包。
- 对于修改资源，比如替换了一张图片之类的情况。我们把它视为新增资源，在打入补丁的时候，代码在引用处也会做相应修改，也就是直接把原来使用旧资源 ID 的地方变为新 ID。

用一张图来说明补丁包的情况，如图 4-1 所示。

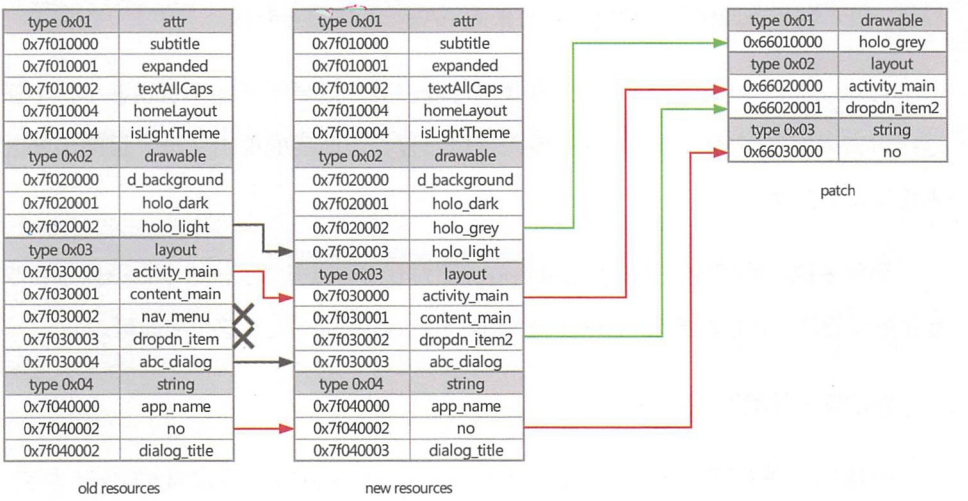


图 4-1 补丁包

图中绿线表示新增资源。红线表示内容发生修改的资源。**黑线**表示内容没有变化，但是 ID 发生改变的资源，× 表示删除了的资源。

4.4.1 新增的资源及其导致的 ID 偏移

可以看到，新的资源包与旧资源包相比，新增了 holo_grey 和 dropdn_item2 资源，新增的资源被加入到补丁包中，并分配了 0x66 开头的资源 ID。

而新增的两个资源导致了在它们所属的 type 中跟在它们之后的资源 id 发生了位移。比如 holo_light，id 由 0x7f020002 变为 0x7f020003，而 abc_dialog 由 0x7f030004 变为 0x7f030003。新资源插入的位置是随机的，这与每次 AAPT 打包时解析 XML 的顺序有关。发生位移的资源不会加入补丁包中，但是在补丁包的代码中会调整 ID 的引用处。

比如说在代码里，我们是这么写的：

```
imageView.setImageResource(R.drawable.holo_light);
```

这个 R.drawable.holo_light 是一个 int 值，它的值是 AAPT 指定的，对于开发者透明，即使点进去，也会直接跳到对应 res/drawable/holo_light.png，无法查看。不过可以用反编译工具，看到它的真实值是 0x7f020002。所以这行代码其实等价于：

```
imageView.setImageResource(0x7f020002);
```

而当打出了一个新包后，对开发者而言，holo_light 的图片内容没变，代码引用处也没变。但是新包里面，同样是这句话，由于新资源的插入导致的 ID 改变，对于 R.drawable.holo_light 的引用已经变成了：

```
imageView.setImageResource(0x7f020003);
```

但实际上这种情况并不属于资源改变，更不属于代码的改变，所以我们在对比新旧代码之前，会把新包里面的这行代码修正回原来的 ID。

```
imageView.setImageResource(0x7f020002);
```

然后进行后续代码的对比。这样后续代码对比时就不会被检测到发生了改变。

4.4.2 内容发生改变的资源

而对于内容发生改变的资源（类型为 layout 的 activity_main，这可能是我们修改了 activity_main.xml 的文件内容。还有类型为 string 的 no，可能是我们修改了这个字符串的值），它们都会被加入到补丁包中，并重新编号为新 ID。

而相应的代码，也会发生改变，比如：

```
setContentView(R.layout.activity_main);
```

实际上也就是：

```
setContentView(0x7f030000);
```

在生成对比新旧代码之前，我们会把新包里面的这行代码变为：

```
setContentView(0x66020000);
```

这样，在进行代码对比时，会使得这行代码所在函数被检测到发生了改变。于是相应的代码修复会在运行时发生，这样就引用到了正确的新内容资源。

4.4.3 删除了的资源

对于删除的资源，不会影响补丁包。

这很好理解，既然资源被删除了，就说明新的代码中也不会用到它，那资源放在那里没人用，就相当于不存在了。

4.4.4 对于 type 的影响

可以看到，由于 type0x01 的所有资源项都没有变化，所以整个 type0x01 资源都没有加入到补丁包中。这也使得后面的 type 的 ID 都往前移了一位。因此 Type String Pool 中的字符串也要进行修正，这样才能使得 0x01 的 type 指向 drawable，而不是原来的 attr。

所以我们可以看到，所谓简单，指的是运行时应用补丁变得简单了。

而真正复杂的地方在于构造补丁。我们需要把新旧两个资源包解开，分别解析其中的 resources.arsc 文件，对比新旧的不同，并将它们重新打成带有新 package id 的新资源包。这里补丁包指定的 package id 只要不是 0x7f 和 0x01 就行，可以是任意 0x7f 以下的数字，我们默认把它指定为 0x66。

构造这样的补丁资源包，需要对整个 resources.arsc 的结构十分了解，要对二进制数形式的一个一个 chunk 进行解析分类，然后再把补丁信息一个一个重新组装成二进制数形式的 chunk。这里面很多工作与 AAPT 做的类似，实际上开发打包工具的时候也是参考了很多 AAPT 和系统加载资源的代码。

4.5 更优雅地替换 AssetManager

对于 Android L 以后的版本，直接在原有 AssetManager 上应用补丁就行了，并且由于用的是原来的 AssetManager，所以原先大量的反射修改替换操作就完全不需要了，大大提高了加载补丁的效率。

但之前提到过，在 Android KK 和以下版本，addAssetPath 是不会加载资源的，必须重新构造一个新的 AssetManager 并加入补丁包中，再换掉原来的。那么我们不就又要和 Instant Run 一样，做一大堆兼容版本和反射替换的工作了吗?

对于这种情况，我们也找到了更优雅的方式，不需要再如此地大费周章。

在 AssetManager 的源码里面，有一个有趣的东西。

文件：frameworks/base/core/java/android/content/res/AssetManager.java。

```
public final class AssetManager {
    ... ...

    private native final void destroy();

    ... ...
```

明显，这个是用来销毁 AssetManager 并释放资源的函数，我们来看看它具体做了什么吧。

```
static void android_content_AssetManager_destroy(JNIEnv* env, jobject clazz)
{
    AssetManager* am = (AssetManager*)
        (env->GetIntField(clazz, gAssetManagerOffsets.mObject));
    ALOGV("Destroying AssetManager %p for Java object %p\n", am, clazz);
    if (am != NULL) {
        delete am;
        env->SetIntField(clazz, gAssetManagerOffsets.mObject, 0);
    }
}
```

可以看到，首先，它析构了 native 层的 AssetManager，然后把 Java 层的 AssetManager 对 native 层的 AssetManager 的引用设为空。

```
AssetManager::~AssetManager(void)
{
    int count = android_atomic_dec(&gCount);
    //ALOGI("Destroying AssetManager in %p #%d\n", this, count);

    delete mConfig;
    delete mResources;

    // don't have a String class yet, so make sure we clean up
    delete[] mLocale;
    delete[] mVendor;
}
```

native 层的 AssetManager 析构函数会析构它的所有成员，这样就会释放之前加载了的资源。

而现在，Java 层的 AssetManager 已经成为了空壳。我们就可以调用它的 init 方法，对它重新进行初始化了！

文件：frameworks/base/core/java/android/content/res/AssetManager.java。

```
public final class AssetManager {
    ... ...

    private native final void init();

    ... ...
```

这同样是个 native 方法：

```
static void android_content_AssetManager_init(JNIEnv* env, jobject clazz)
{
    AssetManager* am = new AssetManager();
    if (am == NULL) {
        jniThrowException(env, "java/lang/OutOfMemoryError", "");
        return;
    }

    am->addDefaultAssets();

    ALOGV("Created AssetManager %p for Java object %p\n", am, clazz);
    env->SetIntField(clazz, gAssetManagerOffsets.mObject, (jint)am);
}
```

这样，在执行 init 的时候，会在 native 层创建一个没有添加过资源，并且 mResources 没有初始化的 AssetManager。然后我们再对它进行 addAssetPath，之后由于 mResource 没有初始化过，就可以正常走到解析 mResources 的逻辑，加载所有此时添加进去的资源了！

文件：android-4.4.4_r2/frameworks/base/libs/androidfw/AssetManager.cpp。

```
const ResTable* AssetManager::getResTable(bool required) const
{
    ResTable* rt = mResources;
    // %% mResources 没有初始化过，为空，因此不会 return
    if (rt) {
        return rt;
    }

     ... ...

    // %% 这时就会走到这里，进行所有 add 进去的 path 的加载
    const size_t N = mAssetPaths.size();
    for (size_t i=0; i<N; i++) {
        // ... 解析 package ...
    }

     ... ...

    return rt;
}
```

这个方案的实现代码如下：

```
        ... ...

        Method initMeth = assetManagerMethod("init");
        Method destroyMeth = assetManagerMethod("destroy");
        Method addAssetPathMeth = assetManagerMethod("addAssetPath", String.
        class);

        // %% 析构 AssetManager
        destroyMeth.invoke(am);

        // %% 重新构造 AssetManager
        initMeth.invoke(am);

        // %% 置空 mStringBlocks
        assetManagerField("mStringBlocks").set(am, null);
```

```
        // %% 重新添加原有 AssetManager 中加载过的资源路径
        for (String path : loadedPaths) {
            LogTool.d(TAG, "pexyResources" + path);
            addAssetPathMeth.invoke(am, path);
        }

        // %% 添加 patch 资源路径
        addAssetPathMeth.invoke(am, patchPath);

        // %% 重新对 mStringBlocks 赋值
        assetManagerMethod("ensureStringBlocks").invoke(am);

    }

    private Method assetManagerMethod(String name, Class<?>...
    parameterTypes) {
        try {
            Method meth = Class.forName("android.content.res.AssetManager")
                              .getDeclaredMethod(name, parameterTypes);
            meth.setAccessible(true);
            return meth;
        } catch (Exception e) {
            LogTool.e(TAG, "assetManagerMethod", e);
            return null;
        }
    }

    private Field assetManagerField(String name) {
        try {
            Field field = mAssetManagerClass.getDeclaredField(name);
            field.setAccessible(true);
            return field;
        } catch (Exception e) {
            LogTool.e(TAG, "assetManagerField", e);
            return null;
        }
    }
```

这里需要注意的地方是 mStringBlocks。它记录了之前加载过的所有资源包的 String Pool，因此很多时候访问字符串是通过它来找到的。如果不进行重新构造，在后面用到它时就会导致崩溃。

由于我们是直接对原有的 AssetManager 进行析构和重构，所有原先对 AssetManager 对象的引用是没有发生改变的，这样，就不需要像 Instant Run 那

样进行烦琐的修改了。

顺带一提，类似 Instant Run 的完整替换资源的方案，在替换 AssetManager 这一步，也可以采用我们这种方式进行替换，省时省力又省心。

4.6 一个意料之外的资源问题

4.6.1 问题的出现

热修复的一个大的挑战在于 Android 系统版本不断升级所带来的不可预知的问题。本节就以资源热修复中遇到的一个问题来说起，详细剖析我们在开发过程中发现和解决问题的过程。

我们之前说过，Sophix 的资源修复，核心思想在于对原来的 AssetManager 注入新的补丁资源。也正是因为采用了这种方式，才避免了新建和替换所有 AssetManager 引用的步骤。我们的资源修复方案在 Android 6.0 之前也一直没有遇到过什么问题，然而某天，在 Android 7.0 版本上，我们遇到了一个诡异的崩溃。

这个崩溃从日志上来看，就是找不到新增的补丁资源。

```
FATAL EXCEPTION: main
Process: com.axon.imoral, PID: 17743
    android.content.res.Resources$NotFoundException: Resource ID #0x66010000
        at android.content.res.ResourcesImpl.getValue(ResourcesImpl.java:255)
        at android.content.res.Resources.loadXmlResourceParser(Resources.
        java:2187)
        at android.content.res.Resources.getLayout(Resources.java:1178)
        at android.view.LayoutInflater.inflate(LayoutInflater.java:424)
        at android.view.LayoutInflater.inflate(LayoutInflater.java:377)
        at someones.Application.activity.MyFragment.onCreateView(MyFragment.
        java:74)
        at android.support.v4.App.Fragment.performCreateView(Fragment.
        java:1962)
        at android.support.v4.App.FragmentManagerImpl.
        moveToState(FragmentManager.java:1067)
        at android.support.v4.App.FragmentManagerImpl.
```

```
        moveToState(FragmentManager.java:1248)
        at android.support.v4.App.BackStackRecord.run(BackStackRecord.
        java:738)
        at android.support.v4.App.FragmentManagerImpl.
        execPendingActions(FragmentManager.java:1613)
        at android.support.v4.App.FragmentManagerImpl.
        executePendingTransactions(FragmentManager.java:570)
        at android.support.v4.App.FragmentPagerAdapter.
        finishUpdate(FragmentPagerAdapter.java:141)
        at android.support.v4.view.ViewPager.populate(ViewPager.java:1106)
        at android.support.v4.view.ViewPager.populate(ViewPager.java:952)
        at android.support.v4.view.ViewPager.onMeasure(ViewPager.java:1474)
        at android.view.View.measure(View.java:19913)
        at android.widget.LinearLayout.measureVertical(LinearLayout.java:928)
        at android.widget.LinearLayout.onMeasure(LinearLayout.java:657)
        at android.view.View.measure(View.java:19913)
        at android.view.ViewGroup.measureChildWithMargins(ViewGroup.
        java:6139)
        at android.widget.FrameLayout.onMeasure(FrameLayout.java:185)
        at android.view.View.measure(View.java:19913)
```

由于我们资源和代码修复是配套地修改，因此，在使用新资源的地方，其代码引用的也是新的资源 ID，也就是以 0x66 开头的资源，如果资源没有打包上，就会出现用这个资源 ID 解析不到资源实体的情况，也就导致了这个问题。

首先，我们能够确定的是，确实有执行资源补丁加载的步骤，然而为什么后面又找不到了？到底后面哪个步骤，把原本已经加载上的资源又弄丢了呢?

这个问题，从 App 的层面是无法进行调查的，因为底层的资源加载，对于上面完全没有感知，所以也就只有在 Android 系统代码中加一些调试信息，重新编译刷机包来复现问题，从而探知底层的行为。

4.6.2 刨根究底

在系统层，所有的资源都是由 ResTable 统一管理，在崩溃之前，我们注意到系统有这一输出。

```
No known package when getting value for resource number 0x66010000
```

查看源码，这是在 ResTable::getResource 函数中打印出来的。

```
ssize_t ResTable::getResource(uint32_t resID, Res_value* outValue, bool
mayBeBag, uint16_t density,
        uint32_t* outSpecFlags, ResTable_config* outConfig) const
{
    if (mError != NO_ERROR) {
        return mError;
    }

    const ssize_t p = getResourcePackageIndex(resID);
    const int t = Res_GETTYPE(resID);
    const int e = Res_GETENTRY(resID);

    if (p &lt; 0) {
        if (Res_GETPACKAGE(resID)+1 == 0) {
            ALOGW(&quot;No package identifier when getting value for resource
            number 0x%08x&quot;, resID);
        } else {
            // 这一行输出错误信息
            ALOGW(&quot;No known package when getting value for resource
            number 0x%08x&quot;, resID);
        }
        return BAD_INDEX;
    }

    ... ...
```

也就是说，资源 ID 为 0x66010000 时无法根据它的 package id 0x66 继续找到具体资源。我们之前已经加载了补丁资源了，而这里却无法找到。那么，这里就可能有两种情况：

1. 加载补丁资源的 ResTable，和找不到资源的 ResTable 是同一个，后面由于某些原因，资源丢失了。

2. 加载补丁资源的 ResTable，和找不到资源的 ResTable 并不是同一个。虽然前者加载了补丁资源，但对后者不产生影响。

要弄清这个问题，我们可以在加载补丁和打印报错信息的时候，输出 ResTable 的 this 指针，只需判断两个 this 的值是不是一样的就可以了。果然，加上日志后，发现两者的 this 确实不同，也就是证实了第二种情况，加载补丁资源的 ResTable，

和找不到资源的 ResTable 并不是同一个。

在系统底层，一个 AssetManager 对应了一个 ResTable，ResTable 不同也就意味着上层是不同的 AssetManager。那么，我们就可以在 AssetManager 的构造函数中输出调用堆栈，看是哪里又生成了新的 AssetManager。

加上日志后，可以看到输出的调用栈是这样的：

```
new AssetManager()

      at android.content.res.AssetManager.&lt;init&gt;(AssetManager.java:99)

      at android.App.ResourcesManager.createAssetManager(ResourcesManager.
      java:246)

      at android.App.ResourcesManager.createResourcesImpl(ResourcesManager.
      java:310)

      at android.App.ResourcesManager.
      findOrCreateResourcesImplForKeyLocked(ResourcesManager.java:345)

      at android.App.ResourcesManager.
      AppendLibAssetForMainAssetPath(ResourcesManager.java:893)

      at android.webkit.WebViewDelegate.addWebViewAssetPath(WebViewDelegate.
      java:205)

      at com.android.webview.chromium.WebViewDelegateFactory$ProxyDelegate.
      addWebViewAssetPath(WebViewDelegateFactory.java:194)

      at com.android.webview.chromium.
      WebViewChromium.&lt;init&gt;(WebViewChromium.java:157)

      at com.android.webview.chromium.WebViewChromiumFactoryProvider.
      createWebView(WebViewChromiumFactoryProvider.java:439)

      at android.webkit.WebView.ensureProviderCreated(WebView.java:2320)

      at android.webkit.WebView.setOverScrollMode(WebView.java:2379)

      at android.view.View.&lt;init&gt;(View.java:4001)

      at android.view.View.&lt;init&gt;(View.java:4118)

      at android.view.ViewGroup.&lt;init&gt;(ViewGroup.java:578)
```

```
    at android.widget.AbsoluteLayout.&lt;init&gt;(AbsoluteLayout.java:55)
    at android.webkit.WebView.&lt;init&gt;(WebView.java:627)
    at android.webkit.WebView.&lt;init&gt;(WebView.java:572)
    at android.webkit.WebView.&lt;init&gt;(WebView.java:555)
    at android.webkit.WebView.&lt;init&gt;(WebView.java:542)
    at android.webkit.WebView.&lt;init&gt;(WebView.java:532)
    at one.third.party.sdk.SDKInterface.&lt;init&gt;(Unknown Source)
    at one.third.party.sdk.SDKInterface.initialize(Unknown Source)
    at someones.App.MyApplication.initBqsSDK(MyApplication.java:579)
    at someones.App.MyApplication.onCreate(MyApplication.java:150)
```

在这里，我们顺藤摸瓜，找到了最初的调用点：

```
    at android.webkit.WebView.&lt;init&gt;(WebView.java:532)
```

似乎是 new WebView 的时候会新建 AssetManager，于是，我们在 7.0 版本的手机上在加载完补丁后，加了一行 WebView 的初始化的代码，果然复现出了这个崩溃，调用栈完全一致。那么，这两者之间为什么会有关联呢？我们就顺着这条调用链继续往下分析，问题的根源也将逐渐浮出水面。

先来看调用链上 addWebViewAssetPath 方法对应的代码：

```
/**
 * Adds the WebView asset path to {@link android.content.res.
   AssetManager}.
 */
public void addWebViewAssetPath(Context context) {
    final String newAssetPath = WebViewFactory.getLoadedPackageInfo()
                                    .ApplicationInfo.sourceDir;

    final ApplicationInfo AppInfo = context.getApplicationInfo();
    final String[] libs = AppInfo.sharedLibraryFiles;
    if (!ArrayUtils.contains(libs, newAssetPath)) {
```

```
            // Build the new library asset path list.
            final int newLibAssetsCount = 1 + (libs != null ? libs.length :
                                                0);
            final String[] newLibAssets = new String[newLibAssetsCount];
            if (libs != null) {
                System.arraycopy(libs, 0, newLibAssets, 0, libs.length);
            }
            newLibAssets[newLibAssetsCount - 1] = newAssetPath;

            // Update the ApplicationInfo object with the new list.
            /* We know this will persist and future Resources created via
               ResourcesManager*/
            /* will include the shared library because this ApplicationInfo
               comes from the*/
            /* underlying LoadedApk in ContextImpl, which does not change
               during the life of the*/
            // Application.
            AppInfo.sharedLibraryFiles = newLibAssets;

            // Update existing Resources with the WebView library.
            ResourcesManager.getInstance().AppendLibAssetForMainAssetPath(
                    AppInfo.getBaseResourcePath(), newAssetPath);
        }
```

首先，WebView 会把自己的资源路径添加到 Resource 中，这时就会调用 ResourcesManager.AppendLibAssetForMainAssetPath。

```
    /**
     * Appends the library asset path to any ResourcesImpl object that
       contains the main
     * assetPath.
     * @param assetPath The main asset path for which to add the library
       asset path.
     * @param libAsset The library asset path to add.
     */
    public void AppendLibAssetForMainAssetPath(String assetPath, String
    libAsset) {
        synchronized (this) {
            // Record which ResourcesImpl need updating
            // (and what ResourcesKey they should update to).
            final ArrayMap&lt;ResourcesImpl, ResourcesKey&gt;
            updatedResourceKeys = new ArrayMap&lt;&gt;();

            final int implCount = mResourceImpls.size();
            for (int i = 0; i &lt; implCount; i++) {
                final ResourcesImpl impl = mResourceImpls.valueAt(i).get();
                final ResourcesKey key = mResourceImpls.keyAt(i);
```

仅供非商业用途或交流学习使用


```
            if (impl != null &amp;amp;&amp;amp; key.mResDir.equals(assetPath)) {
                if (!ArrayUtils.contains(key.mLibDirs, libAsset)) {
                    final int newLibAssetCount = 1 +
                            (key.mLibDirs != null ? key.mLibDirs.length :
                                                    0);
                    final String[] newLibAssets = new String
                                                    [newLibAssetCount];
                    if (key.mLibDirs != null) {
                        System.arraycopy(key.mLibDirs, 0, newLibAssets,
                        0, key.mLibDirs.length);
                    }
                    newLibAssets[newLibAssetCount - 1] = libAsset;

                    updatedResourceKeys.put(impl, new ResourcesKey(
                            key.mResDir,
                            key.mSplitResDirs,
                            key.mOverlayDirs,
                            newLibAssets,
                            key.mDisplayId,
                            key.mOverrideConfiguration,
                            key.mCompatInfo));
                }
            }
        }

        // Bail early if there is no work to do.
        if (updatedResourceKeys.isEmpty()) {
            return;
        }

        // Update any references to ResourcesImpl that require reloading.
        final int resourcesCount = mResourceReferences.size();
        for (int i = 0; i &lt; resourcesCount; i++) {
            final Resources r = mResourceReferences.get(i).get();
            if (r != null) {
                final ResourcesKey key = updatedResourceKeys.get(r
                                                .getImpl());
                if (key != null) {
                    r.setImpl
                    (findOrCreateResourcesImplForKeyLocked(key));
                }
            }
        }

        /* Update any references to ResourcesImpl that require reloading
           for each Activity.*/
        for (ActivityResources activityResources :
             mActivityResourceReferences.values()) {
            final int resCount = activityResources.activityResources
```

仅供非商业用途或交流学习使用


```
                                        .size();
                for (int i = 0; i &lt; resCount; i++) {
                    final Resources r = activityResources.activityResources
                                            .get(i).get();
                    if (r != null) {
                        final ResourcesKey key = updatedResourceKeys.get(r
                                                    .getImpl());
                        if (key != null) {

                       r.setImpl(findOrCreateResourcesImplForKeyLocked(key));
                        }
                    }
                }
            }
        }
```

这时，AppendLibAssetForMainAssetPath 会把传入的 WebView 资源包路径作为 libDirs 参数，构造出新的 ResourcesKey，替换原先还没加入资源路径的 key，并构造出一个更新过 key 的类型为 ArrayMap 的 updatedResourceKeys 变量。这里可看到 libDirs 是直接设置进 mLibDirs 成员的：

```
    public ResourcesKey(@Nullable String resDir,
                        @Nullable String[] splitResDirs,
                        @Nullable String[] overlayDirs,
                        @Nullable String[] libDirs,
                        int displayId,
                        @Nullable Configuration overrideConfig,
                        @Nullable CompatibilityInfo compatInfo) {
        mResDir = resDir;
        mSplitResDirs = splitResDirs;
        mOverlayDirs = overlayDirs;
        mLibDirs = libDirs;
        mDisplayId = displayId;
        mOverrideConfiguration = overrideConfig != null ? overrideConfig :
                                                    Configuration.EMPTY;
        mCompatInfo = compatInfo != null ? compatInfo : CompatibilityInfo
                                        .DEFAULT_COMPATIBILITY_INFO;

        int hash = 17;
        hash = 31 * hash + Objects.hashCode(mResDir);
        hash = 31 * hash + Arrays.hashCode(mSplitResDirs);
        hash = 31 * hash + Arrays.hashCode(mOverlayDirs);
        hash = 31 * hash + Arrays.hashCode(mLibDirs);
        hash = 31 * hash + mDisplayId;
```

仅供非商业用途或交流学习使用


```
        hash = 31 * hash + Objects.hashCode(mOverrideConfiguration);
        hash = 31 * hash + Objects.hashCode(mCompatInfo);
        mHash = hash;
    }
```

AppendLibAssetForMainAssetPath 方法执行到后面，会根据所有 Resource 引用，获取它们的 ResourceImpl，通过之前构造的 updatedResourceKeys 映射表，找到它所对应的 key，并且用 findOrCreateResourcesImplForKeyLocked(key) 重新设置能够与更新后的 key 相对应的 ResourceImpl。

这里的逻辑有点绕，不过我们需要注意的关键点是 findOrCreateResourcesImplForKeyLocked 的实现：

```
    /**
     * Finds a cached ResourcesImpl object that matches the given
       ResourcesKey, or
     * creates a new one and caches it for future use.
     * @param key The key to match.
     * @return a ResourcesImpl object matching the key.
     */
    private @NonNull ResourcesImpl findOrCreateResourcesImplForKeyLocked(
            @NonNull ResourcesKey key) {
        ResourcesImpl impl = findResourcesImplForKeyLocked(key);
        if (impl == null) {
            impl = createResourcesImpl(key);
            mResourceImpls.put(key, new WeakReference&lt;&gt;(impl));
        }
        return impl;
    }
```

可以看到，如果通过这里的 key 无法获取到 ResourcesImpl，就会根据这个 key 创建新的 ResourcesImpl。由于这里的 key 带有 WebView 资源，所以这些已有的 ResourceImpl，它们对应的 key 还是原有，findResourcesImplForKeyLocked 也就无法根据新的 key 找到这些 ResourceImpl。

这也可以从 ResourcesKey 的比较函数中看出：

```
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof ResourcesKey)) {
```

仅供非商业用途或交流学习使用


```
            return false;
        }

        ResourcesKey peer = (ResourcesKey) obj;
        if (mHash != peer.mHash) {
            // If the hashes don&#39;t match, the objects can&#39;t match.
            return false;
        }

        if (!Objects.equals(mResDir, peer.mResDir)) {
            return false;
        }
        if (!Arrays.equals(mSplitResDirs, peer.mSplitResDirs)) {
            return false;
        }
        if (!Arrays.equals(mOverlayDirs, peer.mOverlayDirs)) {
            return false;
        }
        if (!Arrays.equals(mLibDirs, peer.mLibDirs)) {
            return false;
        }
        if (mDisplayId != peer.mDisplayId) {
            return false;
        }
        if (!Objects.equals(mOverrideConfiguration, peer.
            mOverrideConfiguration)) {
            return false;
        }
        if (!Objects.equals(mCompatInfo, peer.mCompatInfo)) {
            return false;
        }
        return true;
    }
```

这里通过对 ResourcesKey 各个字段的严格比较，所以对于 findResourcesImplForKeyLocked 方法：

```
    private ResourcesImpl findResourcesImplForKeyLocked(@NonNull ResourcesKey
    key) {
        WeakReference&lt;ResourcesImpl&gt; weakImplRef = mResourceImpls.
        get(key);
        ResourcesImpl impl = weakImplRef != null ? weakImplRef.get() : null;
        if (impl != null &amp;&amp; impl.getAssets().isUpToDate()) {
            return impl;
        }
        return null;
    }
```

仅供非商业用途或交流学习使用

mResourceImpls.get(key) 就无法找到，所以返回值一定为空。这也就直接运行到了里面 createResourcesImpl 的逻辑中。

```
    private @NonNull ResourcesImpl createResourcesImpl(@NonNull ResourcesKey
    key) {
        final DisplayAdjustments daj = new DisplayAdjustments(key.
        mOverrideConfiguration);
        daj.setCompatibilityInfo(key.mCompatInfo);

        final AssetManager assets = createAssetManager(key);
        final DisplayMetrics dm = getDisplayMetrics(key.mDisplayId, daj);
        final Configuration config = generateConfig(key, dm);
        final ResourcesImpl impl = new ResourcesImpl(assets, dm, config,
        daj);
        if (DEBUG) {
            Slog.d(TAG, &amp;quot;- creating impl=&amp;quot; + impl + &amp;quot; with
            key: &amp;quot; + key);
        }
        return impl;
    }
```

到了这里就豁然开朗了，创建 ResourcesImpl 的时候明确地调用了 createAssetManager 方法来创建新的 AssetManager，并且把新的 ResourcesImpl 回调，这也就间接把老的 ResourceImpl 给丢失了，而老的 ResourceImpl 中，正是包含了我们的补丁资源，因此就在这里也一并被丢弃了。

我们再具体确认一下 createAssetManager 的逻辑。

```
    protected @NonNull AssetManager createAssetManager(@NonNull final
ResourcesKey key) {
        AssetManager assets = new AssetManager();

        /* resDir can be null if the &amp;#39;android&amp;#39; package is creating a
           new Resources object.*/
        /* This is fine, since each AssetManager automatically loads the
           &amp;#39;android&amp;#39; package*/
        // already.
        if (key.mResDir != null) {
            if (assets.addAssetPath(key.mResDir) == 0) {
                throw new Resources.NotFoundException(&amp;quot;failed to add
                asset path &amp;quot; + key.mResDir);
            }
        }
```

仅供非商业用途或交流学习使用


```
        if (key.mSplitResDirs != null) {
            for (final String splitResDir : key.mSplitResDirs) {
                if (assets.addAssetPath(splitResDir) == 0) {
                    throw new Resources.NotFoundException(
                            &amp;quot;failed to add split asset path &amp;quot; +
                            splitResDir);
                }
            }
        }

        if (key.mOverlayDirs != null) {
            for (final String idmAppath : key.mOverlayDirs) {
                assets.addOverlayPath(idmAppath);
            }
        }

        if (key.mLibDirs != null) {
            for (final String libDir : key.mLibDirs) {
                if (libDir.endsWith(&amp;quot;.apk&amp;quot;)) {
                    // Avoid opening files we know do not have resources,
                    // like code-only .jar files.
                    if (assets.addAssetPathAsSharedLibrary(libDir) == 0) {
                        Log.w(TAG, &amp;quot;Asset path &amp;#39;&amp;quot; + libDir +
                                &amp;quot;&amp;#39; does not exist or contains no
                                resources.&amp;quot;);
                    }
                }
            }
        }
        return assets;
    }
```

可见，这里直接创建出一个新的 AssetManager，并且根据传入的 key 的各个字段，逐一把相关资源包通过 addAssetPath 添加进这个新的 AssetManager 中。这里面是不包含任何我们补丁资源的信息的。

至此我们终于弄清楚了原先加入的补丁资源被弄丢的原因了：WebView 的初始化触发了相关资源的注入，因而系统直接构造新的 ResourceImpl，替换掉了原先的 ResourceImpl，而加过补丁资源的 AssetManager 由于是通过 ResourceImpl 进行引用的，也一起被这次替换弄丢了。

仅供非商业用途或交流学习使用

4.6.3 攻克之路

首先我们想到的解决办法是，能不能像 WebView 这种注入资源的方式一样加补丁资源？这样直接利用系统更新机制，可以保证最好的稳定性。

然而尝试过之后发现并不可行，注入的资源无法被找到。因为 WebView 采用的是 AppendLibAssetForMainAssetPath，在 createAssetManager 的时候，最终添加进 AssetManager 时的代码是这样的：

```
protected @NonNull AssetManager createAssetManager(@NonNull final
ResourcesKey key) {
    AssetManager assets = new AssetManager();

    ... ...

    if (key.mLibDirs != null) {
        for (final String libDir : key.mLibDirs) {
            if (libDir.endsWith(&amp;quot;.apk&amp;quot;)) {
                // Avoid opening files we know do not have resources,
                // like code-only .jar files.
                if (assets.addAssetPathAsSharedLibrary(libDir) == 0) {
                    Log.w(TAG, &amp;quot;Asset path &amp;#39;&amp;quot; + libDir +
                            &amp;quot;&amp;#39; does not exist or contains no
                            resources.&amp;quot;);
                }
            }
        }
    }

    ... ...

}
```

因此，最终运行的是 assets.addAssetPathAsSharedLibrary(libDir)，而不是一直使用的 addAssetPath。由于补丁资源和 WebView 不一样，不属于 lib asset，就无法用这种方式来添加。

那么，是否可以在 WebView 触发重构后再次添加补丁资源呢？也就是说，虽然 WebView 重构会导致 AssetManager 重构，但只需要在 WebView 第一次执行初始化后再次添加补丁资源，后面就不会再触发重构逻辑了，因为对应的 key 已经存在

仅供非商业用途或交流学习使用

了，后面所有的 WebView 初始化都没问题。这个方法最为直接了当，我们只需要在 Sophix 里面新增一个添加资源的接口，让用户自行掌握时机触发就行了。但如果站在开发者的角度思考，就会发现这是个很糟糕的方案，因为很多时候开发者并不能确定 WebView 是在哪个时刻发生第一次初始化的，尤其是在大项目中，没有人能够清楚每个细节调用，而且也无法控制项目中的每个开发者以后的代码修改不会影响这个初始化点。还有个问题是，如果开发者调用了某个第三方 SDK，他可能不知道这个 SDK 是否有包含 WebView 的调用。因此，这个方案的可行性是很差的。

当然，我们还可以直接在注入补丁资源之前提前创建 WebView，这样就能避免开发者自己初始化了。但是作为一个负责任的方案提供者，不应该引入任何用户不需要的组件，因为这样会引入额外的开销，并不是所有用户都需要用到 WebView 的。因此，想要使这个问题优雅地解决，还得思考用其他的途径。

于是，我们仔细分析了这段新建 AssetManager 的代码，

```
protected @NonNull AssetManager createAssetManager(@NonNull final
ResourcesKey key) {
    AssetManager assets = new AssetManager();

    ... ...

    if (key.mSplitResDirs != null) {
        for (final String splitResDir : key.mSplitResDirs) {
            if (assets.addAssetPath(splitResDir) == 0) {
                throw new Resources.NotFoundException(
                        &amp;quot;failed to add split asset path &amp;quot; +
                        splitResDir);
            }
        }
    }

    ... ...

}
```

发现了一个可以利用的地方，那就是 mSplitResDirs。

如果能够把补丁资源加载到这里，在重新构建 AssetManager 时，系统就会自

仅供非商业用途或交流学习使用

动把补丁资源添加到新构建的 AssetManager 之中，这样不仅用户没有感知，更不需要过多引入额外的消耗。因此这也是我们最终采用的方案。

具体做法是，反射修改了 LoadedApk 的 mSplitResDirs 字段，加入补丁资源。这样，后面生成 ResourcesKey 的时候就会把相对应的资源设置到自己的 mSplitResDirs 里面。

```
public final class ResourcesKey {
    ... ...

    @Nullable
    public final String[] mSplitResDirs;

    ... ...
```

从而被构造到新的 AssetManager 中，并保留补丁资源。按照这个思路修改完代码，问题不再出现。

4.6.4 不断适应新变化

随着 Android 版本不断更新迭代，系统的各种机制也会随之改变，这种改变对于普通开发者影响不会太大，然而，对于热修复方案的实现却会带来很多的挑战，引入很多之前从未发现的问题。这里只是列举了其中的一个问题，真正工程实践中还需要考虑很多细枝末节的小问题，只有充分考虑清楚，对 Android 系统代码不断精进研究，才能排除所有可能出现的问题。一个好的热修复方案，正是需要不断攻克这类新版本带来的问题，才能与时俱进，永远保持其稳定性和先进性。

仅供非商业用途或交流学习使用

4.7 本章小结

总结一下，相比于目前市面上的资源修复方式，我们提出的资源修复的优势在于：

- **不侵入打包，直接对比新旧资源即可产生补丁资源包（对比修改 AAPT 方式的实现）。**
- **不必下发完整包，补丁包中只包含有变动的资源（对比 Instanat Run、Amigo 等方式的实现）。**
- **不需要在运行时合成完整包。不占用运行时计算和内存资源（对比 Tinker 的实现）。**

唯一有个需要注意的地方就是，因为对新的资源的引用是在新代码中，所有资源修复是需要代码修复的支持的。也因此所有资源修复方案必然是附带代码修复的。而之前提到过，本方案在进行代码修复前，会对资源引用处进行修正。而修正就是需要找到旧的资源 ID，换成新的 ID。查找旧 ID 时是直接对 int 值进行替换，所以会找到 `0x7f??????` 这样的需要替换 ID。但是，如果有开发者使用到了 `0x7f??????` 这样的数字，而它并非资源 ID，可是却和需要替换的 ID 数值相同，这就会导致这个数字被错误地替换。

但这种情况是极为罕见的，因为很少会有人用到这样特殊的数字，并且还需要碰巧这数字和资源 ID 相等才行。即使出现，开发者也可以用拼接的方式绕过这类数字的产生。所以基本可以不用担心这种情况，只是需要注意它的存在。

第 5 章
so 库热修复技术

so 库修复的探索与实践。

仅供非商业用途或交流学习使用

5.1 so 库加载原理

Java API 提供以下两个接口加载一个 so 库。

- `System.loadLibrary(String libName)`：传进去的参数是 so 库名称，表示的 so 库文件，位于 APK 压缩文件中的 libs 目录，最后复制到 APK 安装目录下。
- `System.load(String pathName)`：传进去的参数是 so 库在磁盘中的完整路径，加载一个自定义外部 so 库文件。

上述两种方式加载一个 so 库，实际上最后都调用 `nativeLoad` 这个 native 方法去加载 so 库，这个方法的参数 `fileName：` so 库在磁盘中的完整路径名。

native 方法有动态注册和静态注册两种方式，以下面的代码为例，`stringFromJNI-> Java_com_taobao_jni_MainActivity_stringFromJNI` 静态注册的 native 方法，`test->test` 动态注册的 native 方法。

```
public class MainActivity extends Activity {
    static {
        System.loadLibrary("jnitest");
    }
    public static native String stringFromJNI();
    public static native void test();
}
// 静态注册 stringFromJNI 本地方法
extern "C" jstring Java_com_taobao_jni_MainActivity_stringFromJNI(JNIEnv
*env, jclass clazz) {
```

```
    std::string hello = "jni stringFromJNI old....";
    return env->NewStringUTF(hello.c_str());
}

// 动态注册 test 方法
void test(JNIEnv *env, jclass clazz) {
    LOGD("jni test old....");
}
JNINativeMethod nativeMethods[] = {
        {"test", "()V", (void *) test}
};
#define JNIREG_CLASS "com/taobao/jni/MainActivity"
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved) {
    LOGD("old JNI_OnLoad");
    ... ...
    jclass clz = env->FindClass(JNIREG_CLASS);
    if (env->RegisterNatives(clz, nativeMethods, sizeof(nativeMethods) /
        sizeof(nativeMethods[0])) != JNI_OK) {
        return JNI_ERR;
    }
    return JNI_VERSION_1_4;
}
```

我们知道 JNI 编程中，动态注册的 native 方法必须实现 `JNI_OnLoad` 方法，同时实现一个 `JNINativeMethod[]` 数组，静态注册的 native 方法必须是“`Java_`” + **类完整路径** + **方法名**的格式，如图 5-1 所示。

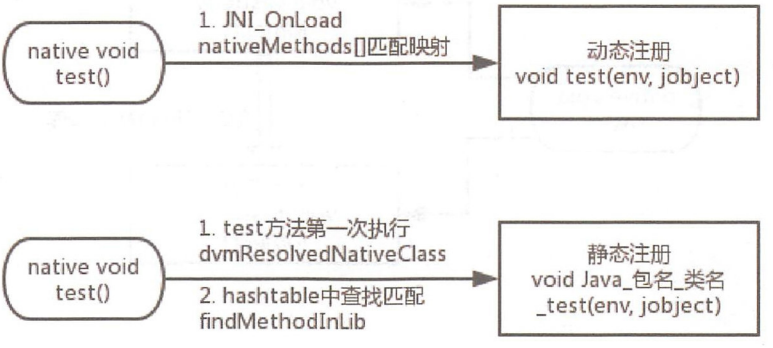


图 5-1 native 方法映射逻辑

总结下：

- 动态注册的 native 方法映射通过加载 so 库过程中调用 `JNI_OnLoad` 方法调用完成。
- 静态注册的 native 方法映射是在**该 `native` 方法第一次执行的时候才完成映射**，当然前提是该 so 库已经加载过。

5.2 so 库热部署实时生效的可行性分析

5.2.1 动态注册 native 方法实时生效

前面分析过 so 库的加载原理，我们知道动态注册的 native 方法调用一次 JNI_OnLoad 方法都会重新完成一次映射，所以我们是否只要先加载原来的 so 库，再加载补丁 so 库，就能完成 Java 层 native 方法到 native 层 patch 后的新方法映射，这样就完成动态注册 native 方法的 patch 实时修复，如图 5-2 所示。

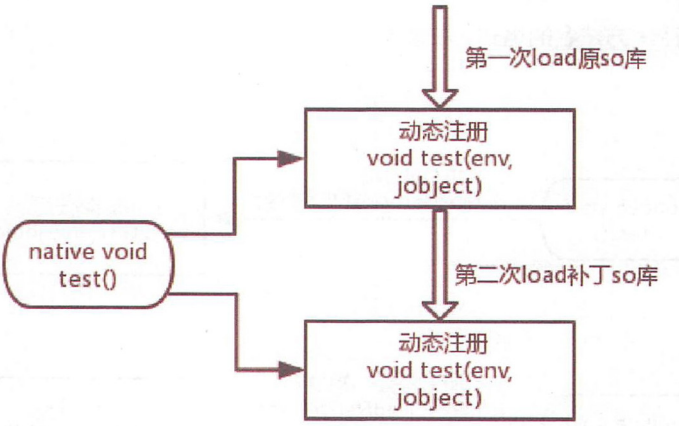


图 5-2 动态注册 native 方法实时生效

实测发现 art 下这样是可以做到实时生效的，但是 Dalvik 下做不到实时生效，通过代码测试我们发现，实际上 Dalvik 下第二次 load 补丁 so 库，执行的仍然是原来

so 库的 JNI_OnLoad 方法，而不是补丁 so 库的 JNI_OnLoad 方法，所以 Dalvik 下做不到实时生效。我们来简单分析下，既然拿到的是原来 so 库的 JNI_OnLoad 方法，那么我们首先怀疑以下两个函数是否有问题。

- dlopen()：返回给我们一个动态链接库的句柄。
- dlsym()：通过一个 dlopen 得到的动态连接库句柄，来查找一个 symbol。

首先来看下 Dalvik 虚拟机下面 dlopen 的实现，源码在 /bionic/linker/dlfcn.cpp 文件，方法调用链路：dlopen → do_dlopen → find_library → find_library_internal。

```
static soinfo* find_library_internal(const char* name) {
  soinfo* si = find_loaded_library(name);
  if (si != NULL) { //so 库已经加载过
    if (si->flags & FLAG_LINKED) {
      return si; // 直接返回该 so 库的句柄
    }
    DL_ERR("OOPS: recursive link to \"%s\"", si->name);
    return NULL;
  }

  TRACE("[ '%s' has not been loaded yet。 Locating...]", name);
  si = load_library(name); //so 库从未加载过，load_library 执行加载
  if (si == NULL) {
    return NULL;
  }
  return si;
}
```

findloadedlibrary 方法判断 name 表示的 so 库是否已经被加载过，如果加载过直接返回之前加载 so 库的句柄，没有加载过，调用 load_library 尝试加载 so 库。

```
static soinfo *find_loaded_library(const char *name) {
    soinfo *si;
    const char *bname;

    // TODO: don't use basename only for determining libraries
    // http://code.google.com/p/android/issues/detail?id=6670
```

```
    bname = strrchr(name, '/');
    bname = bname ? bname + 1 : name;

    for (si = solist; si != NULL; si = si->next) {
        if (!strcmp(bname, si->name)) {
            return si;
        }
    }
    return NULL;
}
```

看代码注释，也知道其实这是 Dalvik 虚拟机下的一个 BUG，这里它是通过 basename 去做查找，传进来的参数 `name` 实际上是 so 库所在磁盘的完整路径，比如此时修复后的 so 库的路径为 `/data/data/com.taobao.jni/files/libnative-lib.so`。但是此时是通过 `bname:libnative-lib.so` 作为 key 去查找的，我们知道第一次加载原来的 so 库 `System.loadLibrary("native-lib");` 实际上已经在 `solist` 表中存在了 `native-lib` 这个 key，所以 Dalvik下面加载修复后的补丁 **so**拿到的还是原 **so** 库文件的句柄，所以执行的仍然是原来 **SO**库的 **JNI_OnLoad** 方法，**Art** 下不存在这个问题，是因为 **Art** 下这个地方是以 **name**作为 **key** 去查找而不是 **bname**，所以 **art** 下重新加载一遍补丁 **so**库，拿到的是补丁 **so**库的句柄，然后执行补丁 **so**库的 **JNI_OnLoad**。

所以如果要解决 Dalvik 下面的这个问题，就需要对补丁中的 **so**进行改名，比如此处补丁 so 库的完整路径修改之后变成 `/data/data/com.taobao.jni/files/libnative-lib-123333.so`，后面一串数字是当前时间戳，确保这个 bname 是全局唯一的，按照上面的分析，在 solist 中查找的 key 已经是唯一的，所以此时可以做到 Dalvik 下面动态注册的 native 方法的实时生效。

5.2.2 静态注册 native 方法的实时生效

上面通过尝试对补丁 so 库进行重命名为全局唯一的名称，可以确保在第二次加载补丁时 so 库可以做到 Dalvik 下和 Art 下动态注册方法的实时生效，而要实现静态注册 native 方法的实时生效还需要更多工作。

前面我们说过静态注册 native 方法的映射是在 native 方法第一次执行的时候就完成了映射，所以如果 native 方法在加载补丁 so 库之前已经执行过了，那么是否这种时候这个静态注册的 native 方法一定得不到修复？幸运的是，系统 JNI API 提供了解注册的接口。

```
static jint UnregisterNatives(JNIEnv* env, jclass jclazz) {
    ClassObject* clazz = (ClassObject*) dvmDecodeIndirectRef(ts.self(),
                                jclazz);
    dvmUnregisterJNINativeMethods(clazz);
    return JNI_OK;
}
/*
 * Un-register all JNI native methods from a class.
 */
void dvmUnregisterJNINativeMethods(ClassObject* clazz){
    unregisterJNINativeMethods(clazz->directMethods, clazz-
        >directMethodCount);
    unregisterJNINativeMethods(clazz->virtualMethods, clazz-
        >virtualMethodCount);
}
static void unregisterJNINativeMethods(Method* methods, size_t count){
    while (count != 0) {
        count--;
        Method* meth = &methods[count];
        if (!dvmIsNativeMethod(meth))
            continue;
        if (dvmIsAbstractMethod(meth))       /* avoid abstract method stubs */
            continue;

        dvmSetNativeFunc(meth, dvmResolveNativeMethod, NULL);
            /*meth->nativeFunc 重新指向 dvmResolveNativeMethod*/
    }
}
```

`UnregisterNatives` 函数会把 `jclazz` 所在类的所有 native 方法都重新指向为 `dvmResolveNativeMethod`，所以调用 UnregisterNatives之后不管是静态注册还是动态注册的 native方法之前是否执行过，在加载补丁 so的时候都会重新去做映射。所以我们只需要以下调用。

```
static void patchNativeMethod(JNIEnv *env, jclass clz) {
    env->UnregisterNatives(clz);
}
```


仅供非商业用途或交流学习使用

这里有一个难点，因为 native 方法的修改是在 so 库中，所以我们的补丁工具很难检测出到底是哪个 Java 类需要解注册 native 方法。这个问题暂且放下。假设我们能知道哪个类需要解注册 native 方法，然后加载补丁 so 库之后，再次执行该 native 方法，这样看起来是可以让该 native 方法实时生效，但是测试发现，在补丁 so 库重命名的前提下，Java 层 native 方法可能映射到原 so 库的方法，也可能映射到补丁 so 库的修复后的新方法。

在静态注册的 native 方法之前从未执行的前提下，首先尝试解析该方法。或者调用了 `unregisterJNINativeMethods` 解注册方法，那么该方法将指向 `meth->nativeFunc = dvmResolveNativeMethod`，真正运行该方法的时候，实际上执行的是 `dvmResolveNativeMethod` 函数。这个函数主要完成 Java 层 native 方法和 native 层方法的映射逻辑。

```
void dvmResolveNativeMethod(const u4* args, JValue* pResult,
    const Method* method, Thread* self) {
    ClassObject* clazz = method->clazz;
    ... ...
    /* now scan any DLLs we have loaded for JNI signatures */
    void* func = lookupSharedLibMethod(method);
        /* 调用 lookupSharedLibMethod 方法，拿到 so 库文件对应的 native 方法函数指针 */
    if (func != NULL) {
        /* found it, point it at the JNI bridge and then call it */
        dvmUseJNIBridge((Method*) method, func);
        (*method->nativeFunc)(args, pResult, method, self);
        return;
    }
    ... ...

    dvmThrowUnsatisfiedLinkError("Native method not found", method);
}

static void* lookupSharedLibMethod(const Method* method){
    return (void*) dvmHashForeach(gDvm.nativeLibs, findMethodInLib,
        (void*) method);
}

int dvmHashForeach(HashTable* pHashTable, HashForeachFunc func, void* arg){
    int i, val, tableSize;
```


仅供非商业用途或交流学习使用

```
    tableSize = pHashTable->tableSize;

    for (i = 0; i < tableSize; i++) {
        HashEntry* pEnt = &pHashTable->pEntries[i];
        if (pEnt->data != NULL && pEnt->data != HASH_TOMBSTONE) {
            val = (*func)(pEnt->data, arg);
            if (val != 0)
                return val;
        }
    }
    return 0;
}
```

gDvm.nativeLibs 是一个全局变量，它是一个 hashtable，存放着整个虚拟机加载 so 库的 SharedLib 结构指针。然后该变量作为参数传递给 dvmHashForeach 函数进行 hashtable 遍历。执行 findMethodInLib 函数看是否找到对应的 native 函数指针，如果第一个找到就直接 return，不再进行下次的查找。

这个结构很重要，在虚拟机中大量使用到了 hashtable 这个数据结构，hashtable 的实现源码在 dalvik/vm/Hash.h 和 dalvik/vm/Hash.cpp 文件中，有兴趣可以自行查看源码，这里不进行详细分析。hashtable 的遍历和插入都是在 dvmHashTableLookup 方法中实现，简单说下 java.hashtable 和 c.hashtable 的异同点：

- 共同点：两者实际上都是**数组实现**，hashtable 容量如果超过默认值都会进行**扩容**，**都是对 key 进行 hash 计算。然后跟 hashtable 的长度进行取模作为 bucket**。
- 不同点：Dalvik 虚拟机下 hashtable put/get 操作实现方法，实际上实现要比 java hashmap 的实现简单一些，java hashmap 的 put 实现需要处理 hash 冲突的情况，一般情况下会通过在**冲突节点上新增一个链表处理冲突，然后 get 实现会遍历这个链表通过 equals 方法比较 value 是否一致进行查找**，davlik 下 hashtable 的 put 实现上 (doAdd=true) 只是简单地把**指针下移直到下一个空节点**。get 实现 (doAdd=false) 首先根据 hash 值计算出 bucket 位置，然后通过 cmpFunc 函数比较值是否一致，不一致，指针下移。

仅供非商业用途或交流学习使用

hashtable 的遍历实际就是数组遍历实现。

知道了 Davlik 下 hashtable 的实现原理，那我们再来看下前面提到的：补丁 so 库重命名的前提下，为什么 Java 层 native 方法可能映射到原 so 库的方法，也可能映射到补丁 so 库的修复后的新方法，如图 5-3 所示。

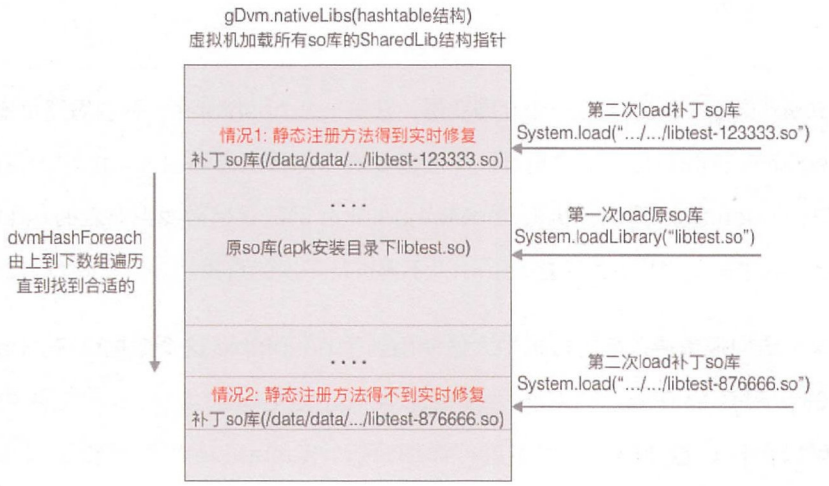


图 5-3 静态注册 native 方法实时生效

所以我们可以得到结论：

对补丁 so 库进行重命名后，如果这个补丁 so 库在 hashtable 中的位置比原 so 库的位置靠前，那么这个静态注册 native 方法就能够得到修复，位置如果靠后就得不到修复。

5.2.3 so 库实时生效方案总结

基于上面的分析，so 库的实时生效必须满足以下几点：

1. so 库为了兼容 Dalvik 虚拟机下动态注册 native 方法的实时生效，必须对 so

仅供非商业用途或交流学习使用

文件进行改名。

2. 针对 so 库静态注册 native 方法的实时生效，首先需要**解注册**静态注册的 native 方法，这个也是难点，因为我们很难知道 so 库中哪几个静态注册的 native 方法发生了变更。假设就算我们知道如果静态注册的 native 方法需要解注册，重新中载补丁 so 库有可能被修复，也有可能不被修复。
3. 上面对补丁 so 进行了第二次加载，那么肯定是多消耗了一次本地内存，如果补丁 so 库够大，补丁 so 库够多，那么 JNI 层的 OOM 也不是没可能。
4. 另外一方面补丁 so 库如果新增了一个动态注册的方法而 dex 中没有相应方法，直接去加载这个补丁 so 文件会报 `NoSuchMethodError` 异常，具体逻辑在 `dvmRegisterJNIMethod` 中。我们知道如果 dex 新增了一个 native 方法，那么就不能热部署只能冷启动重启生效，所以此时补丁 so 库就不能第二次加载了。这种情况下 so 库的修复严重依赖于 dex 的修复方案。

可以看到 so 库实时生效方案，对于静态注册的 native 方法有一定的局限性，不能满足一般的通用性，所以最后我们放弃了 so 库的实时生效需求，转而求其次，实现 so 库修复的冷部署重启生效方案。

5.3 so 库冷部署重启生效实现方案

为了更好的兼容通用性，我们尝试通过冷部署重启生效的角度分析下补丁 so 库的修复方案。

5.3.1 接口调用替换方案

SDK 提供接口替换 System 默认加载 so 库接口：

```
SOPatchManager.loadLibrary(String libName) -> 代替 System.loadLibrary(String libName)
```

`SOPatchManager.loadLibrary` 接口加载 so 库的时候优先尝试去加载 **SDK**

仅供非商业用途或交流学习使用

指定目录下的补丁 so，加载策略如下：

- 如果存在则加载补丁 so 库而不会去加载安装 APK 安装目录下的 so 库。
- 如果不存在补丁 so，那么调用 `System.loadLibrary` 去加载安装 APK 目录下的 so 库。

具体如图 5-4 所示。

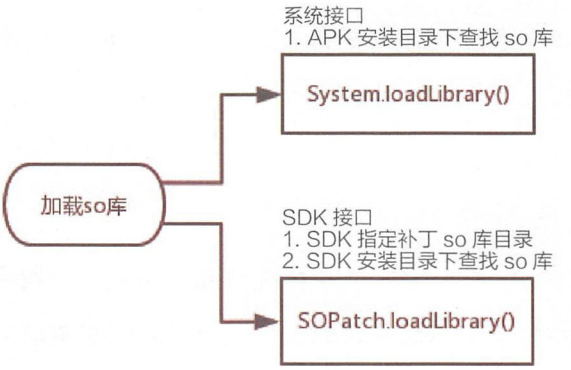


图 5-4　接口调用替换实现

我们可以很清楚地看到这个方案的优缺点。

- 优点：不需要对不同 SDK 版本进行兼容，因为所有的 SDK 版本都有 System.loadLibrary 这个接口。
- 缺点：调用方需要替换掉 System 默认加载 so 库接口为 SDK 提供的接口，如果是已经编译混淆好的三方库的 so 库需要 patch，那么很难做到接口的替换。

虽然这种方案实现简单，同时不需要对不同 SDK 版本区分处理，但是有一定的局限性没法修复三方包的 so 库，同时需要强制侵入接入方接口调用，接着我们来看下反射注入方案。

仅供非商业用途或交流学习使用

5.3.2 反射注入方案

前面介绍过 `System.loadLibrary("native-lib");` 加载 so 库的原理，其实 `native-lib` 这个 so 库最终传给 native 方法执行的参数是 **so 库在磁盘中的完整路径**，比如：`/data/App-lib/com.taobao.jni-2/libnative-lib.so`，so 库会在 `DexPathList.nativeLibraryDirectories/nativeLibraryPathElements` 变量所表示的目录下去遍历搜索。

Android SDK 版本小于 23 时，DexPathList.findLibrary 实现如下：

```
/** List of native library directories。*/
private final File[] nativeLibraryDirectories;
public String findLibrary(String libraryName) {
    String fileName = System.mapLibraryName(libraryName);
    for (File directory : nativeLibraryDirectories) {
        String path = new File(directory, fileName).getPath();
        if (IoUtils.canOpenReadOnly(path)) { //path 文件存在同时
            canOpenReadOnly，返回该 path
            return path;
        }
    }
    return null;
}
```

这里会发现遍历 `nativeLibraryDirectories` 数组，如果找到了 `IoUtils.canOpenReadOnly(path)` 返回为 true，那么就直接返回该 path，`IoUtils.canOpenReadOnly(path)` 返回为 true 的前提肯定是需要 path 表示的 so 文件存在的。那么我们可以采取类似类修复反射注入方式，只要把补丁 so库的路径插入到 nativeLibraryDirectories数组的最前面，就能够使得加载 so库时 加载的是补丁 so 库，而不是原来 so库的目录，从而达到修复的目的。

Android SDK 版本在 23 以上时，DexPathList.findLibrary 实现如下：

```
/** List of native library path elements。*/
private final Element[] nativeLibraryPathElements;
public String findLibrary(String libraryName) {
    String fileName = System.mapLibraryName(libraryName);
    for (Element element : nativeLibraryPathElements) {
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
            String path = element.findNativeLibrary(fileName);
            if (path != null) {
                return path;
            }
        }
        return null;
    }
```

SDK 版本在 23 以上时，findLibrary 实现已经发生了变化，如上所示，那么我们只需要把补丁 so库的完整路径作为参数构建一个 Element对象，然后再插入到 nativeLibraryPathElements数组的最前面就好了。具体如图 5-5 所示。

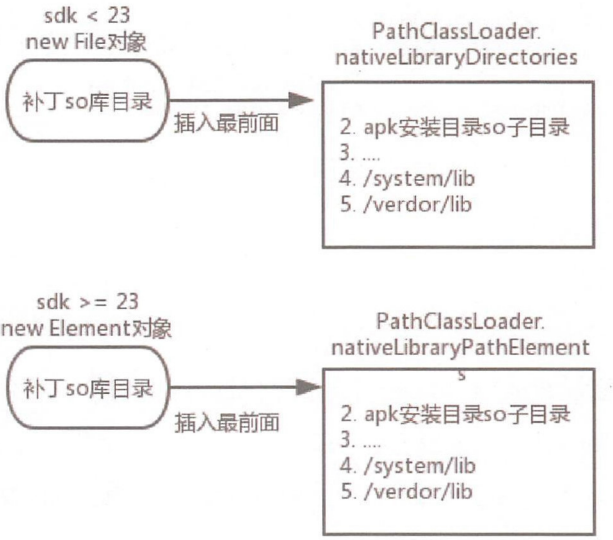


图 5-5　反射注入实现

- 优点：可以修复三方库的 so 库。同时接入方不需要像方案 1 一样强制侵入用户接口调用。
- 缺点：需要不断地对 SDK 进行适配，如上 sdk23 为分界线，findLibrary 接口实现已经发生了变化。

我们知道不管是在补丁包中还是 APK 中，一个 so 库都存在多种 CPU 架构的

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

so 文件，比如“armeabi”“arm64-v8a”“x86”等。加载肯定是加载其中一个 so 库文件的，如何选择机型对应的 so 库文件将是重点所在。

5.4 如何正确复制补丁 so 库

如果在某个机型上包含多个 CPU abi 架构，同时 APK 里面也包含了多个 abi 的 so 库，那么这个 App 在这个机型上运行时空间会选择哪个 abi 的 so 库来执行呢？如图 5-6 所示，简单介绍一下选择 so 的过程。

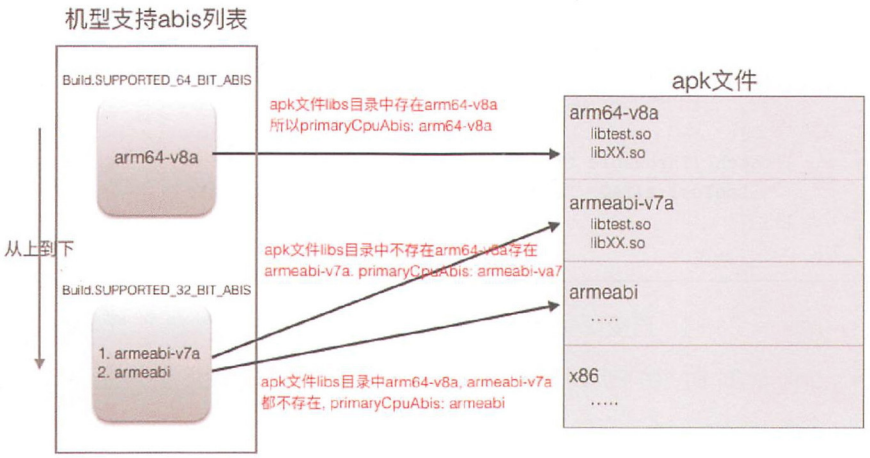


图 5-6 如何选择 primaryCpuAbis

实际上补丁 so 库也存在类似的问题，我们的补丁 so 库文件放到补丁包的 libs 目录下面，`libs` 目录和 `.dex` 文件和 `res` 资源文件一起打包成一个压缩文件作为最后的补丁包，libs 目录可能也包含多种 abis 目录。所以我们需要选择手机最合适的 `primaryCpuAbi`，然后从 libs 目录下面选择这个 primaryCpuAbi 子目录插入到 `nativeLibraryDirectories/nativeLibraryPathElements` 数组中。所以怎么选择 primaryCpuAbi是关键，来看下我们 SDK 具体的实现 。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
static {
    try {
        PackageManager pm = mApp.getPackageManager();
        if (pm != null) {
            ApplicationInfo mAppInfo = pm.getApplicationInfo(mApp
                                            .getPackageName(), 0);
            if (mAppInfo != null) {
                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES
                                                .LOLLIPOP) { //sdk>=21
                    Field thirdFiled = ApplicationInfo.class
                                        .getDeclaredField("primaryCpuAbi");
                    thirdFiled.setAccessible(true);
                    String cpuAbi = (String) thirdFiled.get(mAppInfo);
                    primaryCpuAbis = new String[]{cpuAbi};
                } else { //sdk<21
                    primaryCpuAbis = new String[]{
                            Build.CPU_ABI, Build.CPU_ABI2
                    };
                }
            }
        }
    } catch (Throwable t) {
        LogTool.e(TAG, "SOPatchManager static block", t);
    }
}
```

- sdk ≥ 21 时，直接反射拿到 ApplicationInfo 对象的 `primaryCpuAbi` 即可。
- sdk < 21 时，由于此时不支持 64 位，所以直接把 `Build.CPU_ABI`，`Build.CPU_ABI2` 作为 `primaryCpuAbi` 即可。

5.5 本章小结

对于 so 库的修复方案目前更多采取的是接口调用替换方式，需要强制侵入用户接口调用。目前我们的 so 文件修复方案采取的是反射注入的方案，重启生效，具有更好的普遍性。如果有 so 文件修复实时生效的需求，也是可以做到的，只是有所限制。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 6 章
其他优秀的热修复方案

介绍了 Android 平台上涌现的
其他优秀热修复技术方案。

仅供非商业用途或交流学习使用

6.1 Dexposed 浅析

6.1.1 前言

Dexposed 手机淘宝团队最早使用的热补丁方案，后来开源到 GitHub 上，从其名字 Dexposed，可以看出它是基于大名鼎鼎的 Xposed hook 方案，有饮水思源、回馈开源项目的意思。与 Xposed 不同的是 Xposed 因为它要劫持 Zygote 进程来 hook 系统方法，所以需要 root 权限，但是 Dexposed 是自己 hook 自己的应用，因此不需要 root 权限。

由于目前 Dexposed 一直没能突破 Art 下的限制，Art 确实比 Dalvik 复杂太多；更要命的是，从 Android L 到 Android O，每一个 Android 版本中的 ART 变化都是翻天覆地的，比如方法的内联，混合编译等的引入会导致 hook 艰难无比。同时长江后浪推前浪所以这套方案逐步被放弃，但是这套方案所代表的 hook 思想，却值得我们学习。

6.1.2 Java 函数调用原理

如果你看过 Dalvik 源码，那一定知道在 Dalvik 内部如果需要调用某个函数，则会调用 dvmCallMethod 函数，之后会进行一系列的栈操作。

```
void dvmCallMethod(Thread* self, const Method* method, Object* obj,
    JValue* pResult, ...) {
```

仅供非商业用途或交流学习使用


```
    dvmCallMethodV(self, method, obj, false, pResult, args);
    va_end(args);
}
void dvmCallMethodV(Thread* self, const Method* method, Object* obj,
    ....
    clazz = callPrep(self, method, obj, false); //准备栈帧
      ....
    if (dvmIsNativeMethod(method)) {
        TRACE_METHOD_ENTER(self, method);
        /*
         * Because we leave no space for local variables, "curFrame" points
         * directly at the method arguments.
         */
        (*method->nativeFunc)((u4*)self->interpSave.curFrame, pResult,
                              method, self);
        TRACE_METHOD_EXIT(self, method);
    } else {
        dvmInterpret(self, method, pResult);
    }

#ifndef NDEBUG
bail:
#endif
    dvmPopFrame(self); //出栈
}
```

从上面的代码片段我们可以看到，对于 Java 函数，其处理逻辑由 dvmInterpret 完成，对于 Native 函数，则由对应的 nativeFunc 函数指针完成。

另外一方面，除了 Dalvik 内部函数调用，还有指令比如 invoke-direct，invoke-static 进行函数调用，这里我们来看 invoke-direct 指令是在 portable 模式下是怎么解释的：

```
GOTO_TARGET(invokeStatic, bool methodCallRange)
      ... ...
    GOTO_invokeMethod(methodCallRange, methodToCall, vsrc1, vdst);
GOTO_TARGET_END

#define GOTO_invokeMethod(_methodCallRange, _methodToCall, _vsrc1, _vdst)
goto invokeMethod;

GOTO_TARGET(invokeMethod, bool methodCallRange, const Method* _methodToCall,
    u2 count, u2 regs) {
        ... ...
```

仅供非商业用途或交流学习使用


```
        if (!dvmIsNativeMethod(methodToCall)) {
            ... ...// 省去中间代码，其实类似 dvmCallMethod 逻辑
        } else {
            ... ...
            (*methodToCall->nativeFunc)(newFp, &retval, methodToCall, self);
            ... ...
        }
    }
    assert(false);      // should not get here
GOTO_TARGET_END
```

所以我们可以得到结论，如果一个 Java 方法是 native 方法，那么 Dalvik 内部会跳转到 nativeFunc 函数指针处执行，这个函数指针正是 Dexposed 方案 hook 方法的关键所在。

6.1.3 Dexposed 用法

Dexposed 对于某个函数而言，有三个 hook 点可供选择：函数执行前注入（before），函数执行后注入（after），替换函数执行的代码段（replace）。由此衍生了很多业务层的玩法，典型应用场景如下：

- AOP 编程；
- 插桩（例如测试，性能监控等）；
- 在线热更新，修复严重的，紧急的或者安全性的 BUG；
- SDK hooking 以提供更好的开发体验。

例子 1：应用中所有的 Activity.onCreate(Bundle) 函数调用之前和之后增加一些处理。

```
DexposedBridge.findAndHookMethod(Activity.class, "onCreate", Bundle.class,
new XC_MethodHook() {
        //Activity.onCreate() 之前调用 .
      @Override protected void beforeHookedMethod(MethodHookParam param)
       throws Throwable {
            /* "thisObject" keeps the reference to the instance of target
```

仅供非商业用途或交流学习使用


```
                class.*/
            Activity instance = (Activity) param.thisObject;

            // The array args include all the parameters.
            Bundle bundle = (Bundle) param.args[0];
            Intent intent = new Intent();
            // XposedHelpers provide useful utility methods.
            XposedHelpers.setObjectField(param.thisObject, "mIntent",
            intent);

            /* Calling setResult() will bypass the original method body use
               the result as method return value directly.*/
            if (bundle.containsKey("return"))
                param.setResult(null);
        }

        //Activity.onCreate() 之后调用
        @Override protected void afterHookedMethod(MethodHookParam param)
         throws Throwable {
            XposedHelpers.callMethod(param.thisObject, "sampleMethod", 2);
    }
});
```

6.1.3 Dexposed 源码解析

核心逻辑：在 Dalvik 虚拟机下，主要是通过改变一个方法对象方法在 Dalvik 虚拟机中的定义来实现，具体做法就是将该方法的类型改变为 Native 并且将这个方法的实现链接到一个通用的 Native Dispatch 方法上。这个 Dispatch 方法通过 JNI 回调到 Java 端的一个统一处理方法，最后在统一处理方法中调用 before，after 函数来实现 AOP：

```
    public static XC_MethodHook.Unhook findAndHookMethod(Class<?> clazz,
String methodName, Object... parameterTypesAndCallback) {
        if (parameterTypesAndCallback.length == 0 ||
!(parameterTypesAndCallback[parameterTypesAndCallback.length-1] instanceof
XC_MethodHook))
            throw new IllegalArgumentException("no callback defined");

        XC_MethodHook callback = (XC_MethodHook)
              parameterTypesAndCallback[parameterTypesAndCallback.length-1];
        Method m = XposedHelpers.findMethodExact(clazz, methodName,
                    parameterTypesAndCallback);
        XC_MethodHook.Unhook unhook = hookMethod(m, callback);
```

仅供非商业用途或交流学习使用


```
        if (!(callback instanceof XC_MethodKeepHook
                || callback instanceof XC_MethodKeepReplacement)) {
                   synchronized (allUnhookCallbacks) {
                allUnhookCallbacks.add(unhook);
            }
        }
        return unhook;
    }
```

先调用 XposedHelpers.findMethodExact 找到要 hook 的 Java 方法，再用 hookMethod 进行真正的 hook。

```
    public static XC_MethodHook.Unhook hookMethod(Member hookMethod, XC_
MethodHook callback) {
        if (!(hookMethod instanceof Method) && !(hookMethod instanceof
            Constructor<?>)) {
            throw new IllegalArgumentException("only methods and constructors
            can be hooked");
        }

        boolean newMethod = false;
        CopyOnWriteSortedSet<XC_MethodHook> callbacks;
        synchronized (hookedMethodCallbacks) {
            callbacks = hookedMethodCallbacks.get(hookMethod);
            if (callbacks == null) { // 如果没有修复此方法，就创建一个回调接口的集合
                callbacks = new CopyOnWriteSortedSet<XC_MethodHook>();
                hookedMethodCallbacks.put(hookMethod, callbacks);
                newMethod = true;
            }
        }
        callbacks.add(callback);
        if (newMethod) { // 如果是新方法，获取方法的参数列表和返回值
            Class<?> declaringClass = hookMethod.getDeclaringClass();
            if(runtime == RUNTIME_UNKNOW)  runtime = getRuntime();
            int slot = (runtime == RUNTIME_DALVIK) ? (int)
            getIntField(hookMethod, "slot") : 0;

            Class<?>[] parameterTypes;
            Class<?> returnType;
            if (hookMethod instanceof Method) {
                parameterTypes = ((Method) hookMethod).getParameterTypes();
                returnType = ((Method) hookMethod).getReturnType();
            } else {
                parameterTypes = ((Constructor<?>) hookMethod).
                getParameterTypes();
                returnType = null;
```

仅供非商业用途或交流学习使用


```
            }

            AdditionalHookInfo additionalInfo = new AdditionalHookInfo (callbacks,
            parameterTypes, returnType);
            hookMethodNative(hookMethod, declaringClass, slot, additionalInfo);
        }
        return callback.new Unhook(hookMethod);
    }
```

hookMethodNative 是一个 native 方法，看下这个方法的具体实现：

```
static void com_taobao_android_dexposed_DexposedBridge_
hookMethodNative(JNIEnv* env, jclass clazz, jobject reflectedMethodIndirect,
            jobject declaredClassIndirect, jint slot, jobject
            additionalInfoIndirect) {
    // Usage errors
    if (declaredClassIndirect == NULL || reflectedMethodIndirect == NULL) {
        dvmThrowIllegalArgumentException("method and declaredClass must not
        be null");
        return;
    }

    // Find the internal representation of the method
    ClassObject* declaredClass = (ClassObject*)
        dvmDecodeIndirectRef(dvmThreadSelf(), declaredClassIndirect);
    Method* method = dvmSlotToMethod(declaredClass, slot); /*查找这个 Java 方法
在本地内存中的 Method*/
    if (method == NULL) {
        dvmThrowNoSuchMethodError("could not get internal representation for
        method");
        return;
    }

    if (dexposedIsHooked(method)) { // 判断此方法是否已经被 hook
        // already hooked
        return;
    }

    // Save a copy of the original method and other hook info
    DexposedHookInfo* hookInfo = (DexposedHookInfo*) calloc(1,
        sizeof(DexposedHookInfo));
    memcpy(hookInfo, method, sizeof(hookInfo->originalMethodStruct));
    hookInfo->reflectedMethod = dvmDecodeIndirectRef(dvmThreadSelf(), env->Ne
        wGlobalRef(reflectedMethodIndirect));
    hookInfo->additionalInfo = dvmDecodeIndirectRef(dvmThreadSelf(), env->New
        GlobalRef(additionalInfoIndirect));
```

仅供非商业用途或交流学习使用


```
    // Replace method with our own code
    SET_METHOD_FLAG(method, ACC_NATIVE); // 变更这个方法为 native 方法
    method->nativeFunc = &dexposedCallHandler; //核心逻辑
    method->insns = (const u2*) hookInfo;
    method->registersSize = method->insSize;
    method->outsSize = 0;

    if (PTR_gDvmJit != NULL) {
        // reset JIT cache
        MEMBER_VAL(PTR_gDvmJit, DvmJitGlobals, codeCacheFull) = true;
    }
}
```

`method->nativeFunc = &dexposedCallHandler;` 这里替换了这个方法的 nativeFunc 函数指针，还记得我们在前面一小节，Java 函数调用原理说到，如果该方法是个 native 方法，那么这个方法执行的一定是先执行到 dexposedCallHandler 函数。这里，dexposedCallHandler 函数就是上面说的 Native Dispatch 方法。

```
static void dexposedCallHandler(const u4* args, JValue* pResult, const
Method* method, ::Thread* self) {
    if (!dexposedIsHooked(method)) {
        dvmThrowNoSuchMethodError("could not find Dexposed original method -
            how did you even get here?");
        return;
    }
    ....
    // call the Java handler function
    JValue result;
    //调用了 Java 层的 dexposedHandleHookedMethod 方法
    dvmCallMethod(self, dexposedHandleHookedMethod, NULL, &result,
        originalReflected, (int) original, additionalInfo, thisObject,
        argsArray);
    ....
}
```

dvmCallMethod 前面也介绍过了，Dalvik 内部调用 Java 方法正式使用该方法，dexposedHandleHookedMethod 指针映射到 DexposedBridge 的 handleHookedMethod 方法。

```
private static Object handleHookedMethod(Member method, int originalMethodId,
Object additionalInfoObj,
```

仅供非商业用途或交流学习使用


```
        Object thisObject, Object[] args) throws Throwable {
... ...
// 函数执行前注入(before)
int beforeIdx = 0;
do {
    try {
        ((XC_MethodHook) callbacksSnapshot[beforeIdx]).
         beforeHookedMethod(param);
    } catch (Throwable t) {
        log(t);

        /* reset result (ignoring what the unexpectedly exiting
           callback did)*/
        param.setResult(null);
        param.returnEarly = false;
        continue;
    }

    if (param.returnEarly) {
        /* skip remaining "before" callbacks and corresponding
           "after" callbacks*/
        beforeIdx++;
        break;
    }
} while (++beforeIdx < callbacksLength);

// 执行函数的原本逻辑
if (!param.returnEarly) {
    try {
        param.setResult(invokeOriginalMethodNative (method,
        originalMethodId,
                additionalInfo.parameterTypes, additionalInfo.
        returnType, param.thisObject, param.args));
    } catch (InvocationTargetException e) {
        param.setThrowable(e.getCause());
    }
}

// 函数执行后注入(after)
int afterIdx = beforeIdx - 1;
do {
    Object lastResult =  param.getResult();
    Throwable lastThrowable = param.getThrowable();

    try {
        ((XC_MethodHook) callbacksSnapshot[afterIdx]).
         afterHookedMethod(param);
    } catch (Throwable t) {
```

仅供非商业用途或交流学习使用


```
                DexposedBridge.log(t);

                /* reset to last result (ignoring what the unexpectedly
                   exiting callback did) */
                if (lastThrowable == null)
                    param.setResult(lastResult);
                else
                    param.setThrowable(lastThrowable);
            }
        } while (--afterIdx >= 0);
        ... ...
    }
```

invokeOriginalMethodNative 也是个 native 方法，熟悉 Java 方法到 native 方法映射的同学知道，核心方法就是调用 dvmSetNativeFunc 方法。

```
dvmSetNativeFunc(dexposedInvokeOriginalMethodNative, com_taobao_android_
dexposed_DexposedBridge_invokeOriginalMethodNative, NULL);

static void com_taobao_android_dexposed_DexposedBridge_
invokeOriginalMethodNative(const u4* args, JValue* pResult,
            const Method* method, ::Thread* self) {
    Method* meth = (Method*) args[1];
    if (meth == NULL) {
        meth = dvmGetMethodFromReflectObj((Object*) args[0]);
        if (dexposedIsHooked(meth)) {
            meth = (Method*) meth->insns;
        }
    }
    ArrayObject* params = (ArrayObject*) args[2];
    ClassObject* returnType = (ClassObject*) args[3];
    Object* thisObject = (Object*) args[4]; // null for static methods
    ArrayObject* argList = (ArrayObject*) args[5];

    // invoke the method
    pResult->l = dvmInvokeMethod(thisObject, meth, argList, params,
                    returnType, true);
    return;
}
```

最后通过 dvmInvokeMethod 调用原方法的逻辑，这里省去了很多非核心的代码流程，我们大致上已经明白了 Dexposed 的实现原理。同时这里也存在一个非常不好的体验，就在于 Dexposed 这个动态库的编译必须依赖 AOSP 源码编译，这是

仅供非商业用途或交流学习使用

非常麻烦的。另外一方面由于 Dexposed 始终无法突破 Art 的限制，所以后续这套方案逐渐被抛弃。

6.2 AndFix 探索历程

本节特约作者：黎三平，阿里花名董炼师，Andfix 修复方案主要开发者。

6.2.1 缘起

早在 2013 年的时候，整个业界都在传递着移动互联网时代到来的信号，阿里巴巴当然要引领这个潮流，于是高高地树起了 ALL IN 无线的大旗。此时的支付宝，所有人所有业务无不想着和无线发生点什么反应，于是乎，整个 App 的代码急剧膨胀。是的，要完整地编译出 App 对于开发人员来说也是个难题了，因为喝杯茶的时间已经不可能办到，需要长达一两个小时。

很快地，我们开始了模块化架构，打造出了业界领先的模块化容器。我们可以支持大规模团队的并行开发，每条业务线可以独立迭代研发，当然也能独立发布，是的，我们能够动态发布。所谓动态发布，就是不需要用户升级整个 App，也能够把新的服务、新的 Feature 推送到用户手中。我们把整个 App 做了分层解耦，划分出多个域，每个域下面定义出各个 bundle，每个 bundle 作为部署单元，可以动态发布到终端上。是的，这样很好地支持到团队和业务的扩张，也能够支持业界梦寐以求的动态发布。但是新的问题来了，很多时候我们只是简单的 bug fix，但是发布的时候也需要发布一整个 bundle，甚至多个 bundle，因为我们的 bundle 之间会有依赖。对于用户流量，对于发布的可控上，等等，都是可以继续优化的。

我们很快想到了增量发布，通过 bindiff，再加上一些压缩，能够不错地降低流量的成本。但是，我们做 bug fix 的时候，很多时候只是几行代码的修改，一个 crash，很多情况下就是少个非空判断，或者少了个同步的锁。我们就去思考能不能去轻量地

仅供非商业用途或交流学习使用

解决这个问题，其实我们只需要下发那几行修改过的代码，让它能够加载执行就可以了。关键的是要对开发者友好，尽量对开发者透明。

6.2.2 开始探索

对于用 Windows 的用户，一定对于安装升级补丁包印象深刻。用户可以在更新和安全中检测更新，下载安装类似 KBXXXXXX 的补丁，有些安装需要重启操作系统，有些安装不需要重启操作系统，总之，肯定不是要让用户重新安装个新的操作系统。那么对于移动应用是不是也可以打补丁呢？对于这个命题，需要回答几个问题：

1. 补丁包怎么加载并运行起来？

2. 补丁包怎么来？

3. 补丁包怎么安全地下发到用户手中？

很显然，要让“泼出去的水”发布出去的版本能够通过云端来做 bug fix，这些问题都是需要解决的，缺一不可。一个名叫 AndFix 的项目开始悄悄上路，它的命名源于 Android-HotFix 的缩写。

6.2.3 补丁包的加载运行

要让一个补丁包加载并运行起来，必然是所有命题中的一个核心命题。对于移动应用来说，它不像操作系统那样一般长时间运行，可以在用户下次启动应用的时候更新程序就好了。另外一种比较酷的方式就是“空中换引擎”，在程序运行的过程中更新程序。

对于一个超级应用（Super App）来说，用户使用我们这个应用的姿势是难以预测的，特别是业务上在做一些大的运营活动时，比如当所有用户在开着 App 准备抢红包时，这个时候如果发现程序有个缺陷，该怎么办？嗯，很有可能发生大的资损，对于 Server 端来说，我们最差也可以做一次紧急发布，但是对于 App 端呢？我们唯

仅供非商业用途或交流学习使用

有在程序的执行过程中“空中换引擎”。

如果一个 hacker 要去 hack 一个运行中程序，他需要先注入到这个进程，把自己的代码挂载上去，修改程序指针……此间会有很多问题要解决，首先要解决的就是权限的问题，怎么对内存执行写操作呢？怎么让一块内存可执行呢？回到 Android 上面来，我们开发应用主要用的是 Java 语言，程序是运行在 Dalvik 或者 Art 虚拟机上，它有几个特点：

1. 托管代码的方式，如果站在虚拟机的层面，是有权限去修改虚拟机字节码所占的内存的。

2. 对于当前进程，可以很轻松地挂载动态的可执行库，无论是用 dyld，还是说就用 Java 的 load。

3. 可以比较方便地开辟出一块新的字节码的内存，并设为可执行的权限，是的，通过 ClassLoader 动态加载一个类就搞定了。

4. 程序第一次加载前一般会对字节码做一次优化，对于 Dalvik 是 dexopt，对于 Art 就是 dex2oat。

对于第 1 点，修改虚拟机字节码所占的内存对于整体方案来说复杂度还是比较高的，主要是内存管理方面，需要把新的字节码给覆盖掉原来的内存空间，这个在实际操作中还是比较复杂的。对于第 4 点，如果在加载执行之前对字节码做好修改，再合并成一个完整的字节码，但是对于字节码的优化还是比较耗时的，而且对于 CPU 的加载和内存也是有影响的。

一般虚拟机在运行的时候，加载字节码之后就会根据字节码构造相应的语法结构，所以是否可以在这个地方切入呢？答案是肯定的。如果我们把内存中 Method 对应的内存结构做一下修改，把其中的指令语句的内存 offset 换成其他指令语句的地址是不是就搞定了，嗯，切入点已经找到，接下来就是逐个闯关吧。

仅供非商业用途或交流学习使用

6.2.4 方法替换

方向基本确定了，我们就是要在运行时对方法做些指令替换，以达到我们补丁包加载执行的目的。为了让整个方案足够简单，足够轻量，所以没有考虑在运行时动态生成字节码，因为这不管是方案上，还是性能上都会有一定的挑战。另外我们前面也提到，目的只是为了做 bug fix，所以只要能够达到对方法做简单的修改即可。我们巧妙地利用现有的机制，直接加载事先准备好的字节码，把它们的相应数据结构做下替换。简单地说就是在运行的时候，用新加载的一个方法来替换掉现有的方法，也就是方法替换，如图 6-1 所示。因为仅仅是方法替换，所以对外的接口需要保持一致，也就是说需要替换和被替换的两个方法的签名是一致的，包括方法名、入参、出参，以及相关的类型，这对于来做 bug fix 的已经足够。

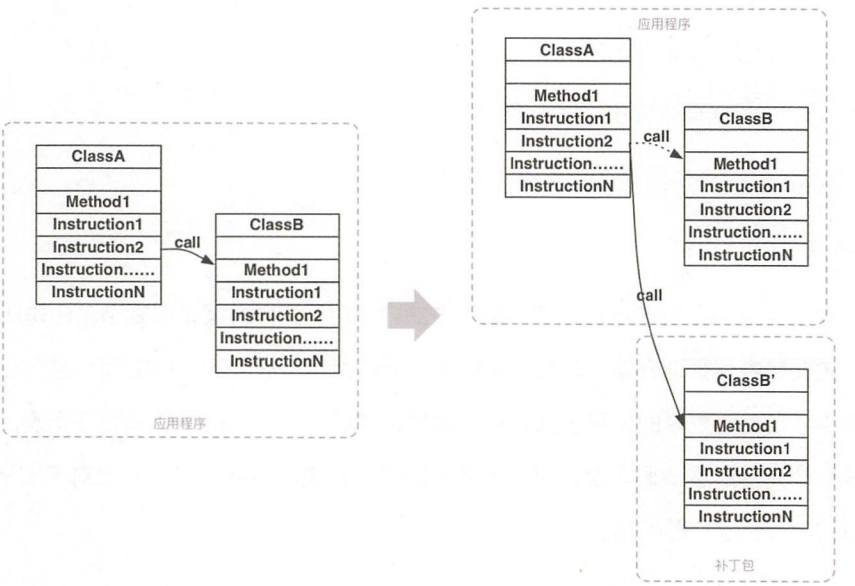


图 6-1 方法替换

但是接下来，还是有很多疑难杂症需要去解决，主要是 verify 的问题：

1. 类的加载问题。在 Java 语言规范中，Java 的类加载模型是双亲委派的，一个类依赖的类需要在当前类的 ClassLoader 或者当前类的 ClassLoader 的

仅供非商业用途或交流学习使用

ParentClassLoader，或者依此类推的 ClassLoader 中能够加载进来，否则就会出现类加载的异常。那么在我们的方案中，直接来加载事先准备好的字节码，但是它所依赖的类还在原来的 ClassLoader，所以要么用来加载的这个 ClassLoader，它的 ParentClassLoader 是依赖的类的 ClassLoader，要么就需要打破这个双亲委派模型。如果这个 ClassLoader 的 ParentClassLoder 是依赖的类的 ClassLoader，那么对于类名也有要求，因为在一个类加载的链路上，一个类名只可以唯一地对应到一个类，不可以多个类的名字相同，否则就要看加载顺序了，如图 6-2 所示。另外对于新加载的类的 ClassLoader 我们也需要改成原类型所对应的 ClassLoader，否则在方法中所依赖的类的加载也将成为一个问题。

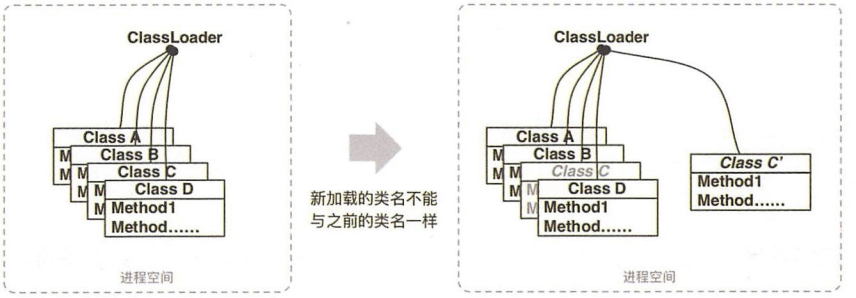


图 6-2 类加载顺序

2. 对于方法所在类的校验。对于实例方法，它的 invoke 的第一个参数就是实例本身，但是实例的类型却不是之前方法所在的类的类型了，它的类型是类加载的时候所在的类的类型。所以我们需要去想办法绕过这个限制，有一种做法就是把新加载的类的类型作为原类的子类型，不过这种办法需要去填的坑一定不少；另一个简单的办法就是把替换过来的方法的类指向原来的类，我们采取的方式是后者，当然也有一些坑需要去填。

3. 对于方法调用的权限校验。我们知道，对于 private 的方法，在类的外部是没法调用的，这个虚拟机层面会做这样校验。我们的解决办法也很简单粗暴，直接把方法的修饰改成 public 的。

仅供非商业用途或交流学习使用

实施上，就是创建一个以程序当前 ClassLoader 为 ParentClassLoader 的 ClassLoader 来加载补丁类，补丁类不与原程序的类同名；再在虚拟机层面挂载方法替换的动态库，我们采用的就是简单的 Java 来 load，然后通过 JNI 来做虚拟机上的操作，比如把类或者方法转成内存地址，并对相应的抽象语法树层面的数据结构做操作，把对应 Method 的结构做相应的修改及替换。最终达到在运行时作方法替换的目的。

6.2.5 补丁包的制作

现在在运行时能够做方法替换了，需要去提供做方法替换所需的目标方法，也就是说我们需要给出补丁包。对于补丁包的制作，我们提了两点要求：

1. 补丁包一般是由开发者开发出来的，所以我们一定要做到开发者友好。

2. 补丁包要足够小，这样不管是对于用户流量，还是对于实时触达都是大有裨益的。

如果熟悉 Linux 的话，在 Linux 上面会很容易接触到 patch，比如要编译安装一个软件，常常会先下载一个软件的源代码，然后为了得到比较新的版本，又会在社区下载对应的 patch 包，之后打上这个补丁再编译安装。是的，这种补丁其实就是源代码的 diff 包，我们想要的是像 Windows 更新的那种补丁包，编译完成可直接执行的字节码。所以，我们综合这两者的优点，开发者只需要走正常的迭代那样去拉分支做 bug fix 就好了，编译构建出一个稳定的程序包版本，然后跟老版本的程序包做 diff，产生一个可执行的补丁包。我们定义的 bug fix 的研发流程会是这样的，如图 6-3 所示。

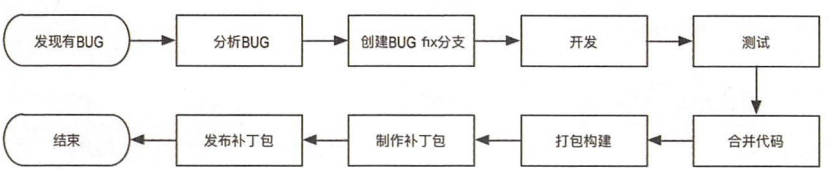


图 6-3 bug fix 研发流程

仅供非商业用途或交流学习使用

现在就差一个制作补丁包的工具，这个制作补丁包的工具需要可以对可执行程序包做 diff，也就是说能对 APK 文件做 diff。有一种选择就是前面讲到的 bindiff，但是我们还是希望能从代码的逻辑层面来做 diff。

我们开发了一个这样的工具，对 APK 文件中的字节码做分析，构造出抽象语法树，然后逐个做 diff，发现其中有不一样的地方就对该方法做个标记，然后在最后把所有不一样方法的类重新编译成一个字节码文件，扩展名为 apatch（名称取自 Android patch 的缩写）。期间这个工具还做了一些其他工作。

1. 对有变更的方法做标记。我们就是在方法上面加了一个运行时的注解，当补丁包被加载的时候，会遍历所有包含该注解的方法，调用方法替换的动态库执行方法替换。主要目的有两个：一个是在运行时对方法做替换能够做到可控，另一个是性能，加快替换速度，把不需要替换的方法给甄别出来。

2. 对类名做更改。前面有讲到类加载不可以加载同名的类，而我们做 diff 的时候，都是基于同名的类来做的，所以之后打包需要修改类名以确保运行的时候 ClassLoader 能正确地加载。我们改名逻辑也比较简单，就是在原类名之后加个后缀，后缀其实就是一个固定的魔法串再加上时间戳。之所以加上时间戳，目的就是为了把它当做一个版本号，因为运行起来的同时是可能存在多个补丁版本并行的。另外我们的一个设计哲学就是，补丁包永远向前走，没有回滚，要做回滚只能是把之前的包跟当前的包做个 diff，带上新的时间戳（版本）再发布一次。

3. 对补丁包签名。确保补丁包不会被篡改，不管是网络传输中，还是在存储中。

另外，对于补丁包的制作需要去解决其他的一些问题，比如：

1. 对于混淆的支持。对于使用 proguard 来做代码混淆的，这个比较简单，因为 proguard 本身提供保存字典的，并可以利用之前保存的字典来做新版本的代码混淆。开发者需要做的是保留每个版本的 mApping 文件，不过一般应该都有保存，否则出现问题了怎么去做符号解析来辅助问题排查呢。构建新版本使用该 mApping 文件做

仅供非商业用途或交流学习使用

输入，依此类推。

2. 对于加固的支持。目前加固方式琳琅满目，原理也各不相同，不过总可以保留加固之前的原始包吧，制作补丁包的时候就使用原始包来做输入。

6.2.6 补丁包的安全下发

补丁包制作完成之后，需要安全地下发到用户手中。下发这块的能力是需要后台系统支持的，我们是有原有的系统支持的，而且能力上也算是比较成熟，下发策略方面支持灰度发布，通道方面既支持推送也支持拉取，包括按照不同网络环境等条件来做下发，这块就不做过多地阐述了。在补丁包的安全方面，我们做了以下两点：

1. 前面提到的在制作补丁包的时候做签名。目的主要是防篡改，我们会在应用程序中去取公钥，在补丁包每次加载的时候做验证，原理上其实就类似 apk 的签名验证。

2. 会对优化后的字节码文件做摘要。目的主要是防止对优化后的文件做篡改，就是为应对“寄生兽”的漏洞，对于这块的处理其实很简单，只是简单地做下 MD5 摘要并对摘要存储，之后每次加载的时候都去做这个验证。

6.2.7 局限性及未来

AndFix 在设计和落地实现的时候就决定了它是有一些局限性的，主要是如下两个方面：

1. 只能要来做简单的 bug fix。首先，设计之初要解决的问题不是动态发布，因为动态发布我们已经有别的方式来解决了，所以对于 AndFix 的定位仅仅是为了解决版本发布之后比较棘手的问题，在方案上面就设计成方法替换；其次，我们认为对于资源这块所能产生问题的影响不大，而且我们的场景对于 Native 的 so 库的热修复也不多，另外也有冷启动之后再重新加载新的版本 so 库的能力，所以这些方面没有做

过多的考虑，主要还是希望 AndFix 能够做到足够的简单，足够的轻量，不管是对于开发者，还是对于用户的流量。

2. 兼容性的问题。首先，在方案确定之初就想着从虚拟机底层去做些突破，这块其实或多或少地有些“黑科技”的味道，走的就是 Hack 的方式；其次，在实现的过程中，就同时还在进行着 Hook 的研究，希望在调用完 Hook 的方法还能调回来，回到原来的方法去执行，目前阿里巴巴已经有同事基于此完成了整个 Hook 方案的实现。所以在实现过程中有对各个版本的虚拟机数据结构做版本兼容，主要还是由于 Hook 需要感知这些差别，而没有直接去走内存 Copy 的方式，当然走内存 Copy 的方式更为简单。

至此，在这个移动开发百花齐放，百家争鸣的时代，各种热修复技术层出不穷，AndFix 作为早期的 Android 热修复解决方案，也必将淹没在这滚滚浪潮之中。

作为一个很早就是较完整的项目来面向开发者开源，我们谨希望 AndFix 的系统化解题思路和实现方案，能够给开发者们带来一些新思路、新思维！

6.3 Amigo 核心解读

本节特约作者：曹玉斌，阿里花名夜沧，Amigo 热修复方案主要开发者。

回想 Amigo 开源一年多以来，笔者从饿了么转战蚂蚁金服，几个核心 Amigo 小伙伴也继续在其他一线互联网企业奋战，虽天各一方，但对技术的追求不止。借此机会，深度剖析一下 Amigo 的核心实现之路。

6.3.1 为什么要做 Amigo

Amigo 诞生于饿了么移动组，在饿了么业务快速成长期笔者加入了这家公司，那是在 2014 年年底。在随后一年的业务高速发展期，移动组的全体同学把所有精力

都放在解决业务上的需求。随后 2017 年业务趋于稳定，团队也加入了不少新同学，这个时候对于代码质量的把控愈显力不从心。记得有段时间每个 PR 的合入，都要将近十名开发者复看代码，那个场景基本上就是一个同学写完了代码，一直追在 N 个同学后面求复看代码。在业务压力下，很难保证代码的质量。

经过了自测、QA 测试之后，线上出了问题怎么办？一般先看影响范围，针对不同级别的线上问题，有不同的技术预案。接口数据上能暂时解决的，优先让后端的同学出来顶锅，对某个平台定点适配并上线；不是连续闪退、业务不可用的时候，顺延到下个版本；出现连续闪退、业务不可用的时候，此时只能紧急发版了（在灰度发布下，这种情况极少出现）。

为什么要做 Amigo？有没有成熟解决方案？技术同学是得了懒癌的一群人，如果有稳定的方案，拿来主义是所有人都喜闻乐见的。很遗憾，当时并没有稳定通用方案。

6.3.2 Amigo 代号由来

当时笔者在着手饿了么热修复框架的搭建，写了一部分内容，但是方案名还没有想好。一次饭间我把手头的上的事和大家说了一下，这时 Amigo 从一位同学的口中脱口而出。amigo [əˈmiːgəu]，源自西班牙语，朋友的意思。热睡复技术本就是充当线上救火的角色，正如遇到困难时来自朋友的一臂之力，以此为名也算贴切。

6.3.3 feature 及实现原理

Amigo 的核心思想是借用 App 的运行时，偷梁换柱替换整个 App 运行的代码，包括 dex、so、assets。Amigo 接收的 patch 其实是一个可独立安装运行的 APK，替换的是整个运行时代码，所以也不会存在 Dalvik 和 ART 下类似 class pre-verified 问题，也就不需要像其他方案一样进行插桩。具体其中的逻辑包括了如何替换 App Application、Amigo 自修复、修复类、修复 so、修复资源、patch 下

发，以及 ROM 兼容性这几块。

6.3.4 Application 替换

出于集成便利性的考量，无 / 低成本接入是必要的。Android 项目大多是基于 gradle 进行依赖、编译、达到的，Amigo 的做法是输出一个 gradle plugin，无缝的对接已有项目。开发者只需一行申明语句引入插件即可完成。

```
buildscript {

    dependencies {
        classpath 'me.ele:amigo:x.x.x'
    }
}

Apply plugin: 'me.ele.amigo'
```

Android 编译打包其中有一步是合并处理 manifest---processManifest，Amigo plugin hook 了这一 task。在 processManifest 结束之后，生成的一个 AndroidManifest.xml，路径如下 ../build/build/intermediates/manifests/full/release/AndroidManifest.xml。这里面包含了 App 所有的配置信息，Application 是四大组件的 root node，对应的 name 属性值是 App 的入口类。Amigo 替换掉 name 属性值，这样就可以掌控住 App 的入口，完成 patch 包的加载等逻辑，之后把入口逻辑再转发到 App 原有的入口类中去。这种处理方式其实很普遍地应用在 App 加固领域，Amigo 借鉴了这种思想应用在热修中，同时也满足了修复 Application 的需求。

Amigo Plugin 流程如图 6-4 所示。

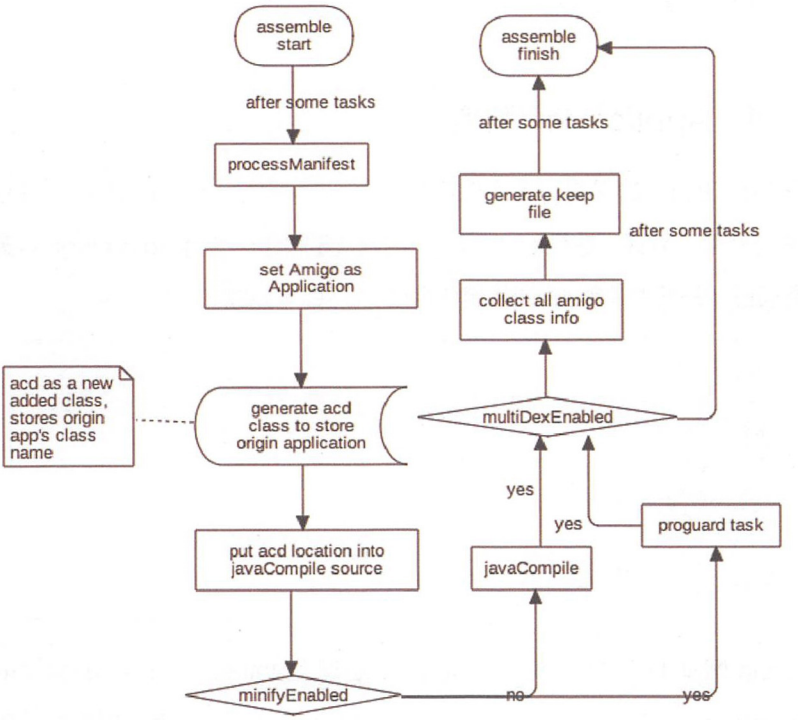


图 6-4　Amigo Plugin 流程

6.3.5　Amigo 自修复

Amigo 本质上是一个 gradle plugin 和一个 Android library。plugin 是本地依赖的，如有 BUG，是可以在本地及时修改的，不会带到线上去。library 是会打到 APK 中，运行时执行的，是代码就可能会 BUG，所以 Amigo 也支持自我热修复。入口类的入口方法，比如 attachBaseContext(Context base)、onCreate() 逻辑是不好修复的。因为这是程序最开始执行的地方，所幸这里的逻辑比较简单，出现 BUG 的可能性也比较小。真正自我修复是内部的逻辑，包括释放 APK，校验、加载 dex/so 文件等，这些才是关键的，才是程序执行的核心。

实现的思想是检测到 patch 存在的情况下，直接加载 patch 包中 Amigo 相关类，保证执行的逻辑是最新的。

6.3.6 修复 class

修复 class 的解析文章比较多，这里不做深度剖析，在此说些 Amigo 独有的东西。我们知道 App 的类加载器是 PathClassLoader，所有类的查找、加载都在这里完成，其他方案的思路基本是参考 MultiDex 一层层反射去操作 DexPathList 中的 dexElements。Amigo 处理策略会有所不同，因为 patch 是一个完整全新的 APK，解压 dex 完毕后，先预先对所有的 dex 文件进行 dexopt/dex2oat 操作，然后重新构建一个 ClassLoader。这样做的好处有两个：一是会加快启动的速度；另一是不需要大量的反射处理提升了兼容性。

```
// snippet in AmigoClassLoader
public static AmigoClassLoader newInstance(Context context, String checksum)
{
        String apkPath = PatchApks.getInstance(context).patchPath(checksum);
        String dexPath = getDexPath(context, checksum);
        String dexOptDir = AmigoDirs.getInstance(context).dexOptDir(checksum)
                                .getAbsolutePath();
        String libPath = getLibraryPath(context, checksum);
        ClassLoader parent = AmigoClassLoader.class.getClassLoader()
                                .getParent();
      return new AmigoClassLoader(apkPath, dexPath, dexOptDir, libPath,
             parent);
}

// snippet  in Amigo
private static boolean checkAndSetAmigoClassLoader(Context context) {
        try {
            String classloaderName = context.getClassLoader().getClass()
                                        .getName();
            if (classloaderName.equals(AmigoClassLoader.class.getName())) {
                return false;
            }
            Context App = context.getApplicationContext();
            ClassLoader classLoader = App.getClass().getClassLoader();
            writeField(getLoadedApk(), "mClassLoader", classLoader);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
        }

        return false;
}
```

6.3.7 修复 Application、四大组件

Application 和四大组件也是 class，只不过继承了 Android framework 中一些基类，并把它们声明在了 AndroidManifest 中。AndroidManifest 文件中的四大组件是被系统识别的，四大组件的管理也是由系统完成的。如果是类，就可以被修复，上一步已经介绍得比较清楚了，不再赘述。需要额外提一下的是 Application 作为 App 的入口类是如何被修复的呢，前面介绍 Amigo Plugin 的一个 task 就是在构建的时候把 Application 给替换掉，保证了在热修复逻辑走完之后才会去加载原来的 Application。

如果有新的四大组件加入，如何保证组件能被正常启动呢？没有解决方案是通过 Delegate 完成的，Activity 通过 hook 加载流程，打开对应 launch mode 的 StubActivity，再反射注入新 Activity 的实例；Service、Provider 也是通过 ServiceStub、ProviderStub 实现的；BroadcastReceiver 是变静态注册为动态注册实现的。

此处以 Activity 的启动时序图为例展开描述，如图 6-5 所示。

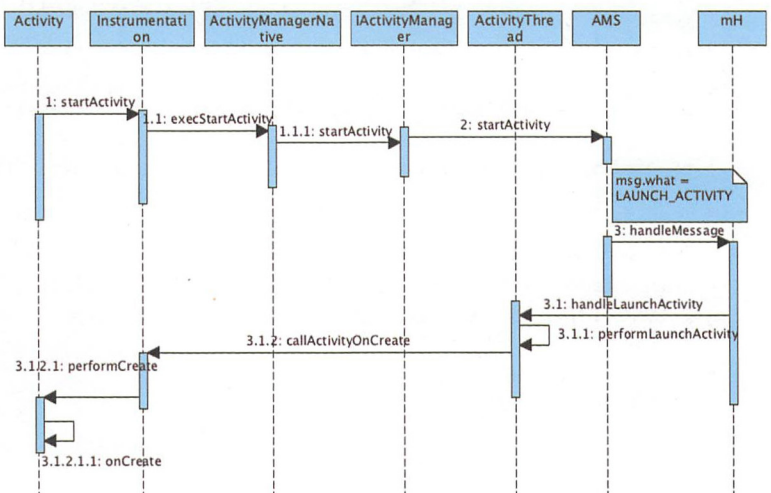


图 6-5 Activity 启动时序

在 patch apk 里面的新增 Activity，是不能被系统正常启动的，因为系统在查看 AndroidManifest 并没有注册当前 Activity，就会抛异常出来。

```
// snippet  in Instumentation
public static void checkStartActivityResult(int res, Object intent) {
                ...
        switch (res) {
            case ActivityManager.START_INTENT_NOT_RESOLVED:
            case ActivityManager.START_CLASS_NOT_FOUND:
                if (intent instanceof Intent && ((Intent)intent).
                    getComponent() != null)
                    throw new ActivityNotFoundException(
                            "Unable to find explicit activity class "
                            + ((Intent)intent).getComponent().toShortString()
                            + "; have you declared this activity in your
                               AndroidManifest.xml?");
                throw new ActivityNotFoundException(
                        "No Activity found to handle " + intent);
```

Amigo 采用的做法是“偷梁换柱”，向系统发起 startActivity 请求的时候是打开一个占位页面 (StubActivity)，hook Instrumentation.execStartActivity(..) 跳转。

```
// snippet  in AmigoInstumentation
private Intent wrapIntent(Context who, Intent intent) {
       ... ...
        ComponentName componentName = intent.getComponent();
        ActivityStub.recycleActivityStub(getActivityInfo(who, componentName.
            getClassName()));
        Class stubClazz = getDelegateActivityName(who, componentName
                              .getClassName());
        if (stubClazz == null) {
            Log.e(TAG, "wrapIntent: weird, no stubs available for now.");
            return intent;
        }

        Intent stubIntent = new Intent();
        stubIntent.setComponent(new ComponentName(componentName.
            getPackageName(), stubClazz.getName()));
        stubIntent.putExtra(EXTRA_TARGET_INTENT, intent);
        stubIntent.setFlags(intent.getFlags());
        intent.putExtra(EXTRA_STUB_NAME, stubClazz);
        ActivityStub.onActivityCreated(stubClazz, null, componentName.
            getClassName());
        return stubIntent;
    }
```


仅供非商业用途或交流学习使用

第二步，hook mH.handleMessage(..)，当占位页面的回调回来的时候，再在当前 App 进程空间内部把 message 中 activityInfo 和 intent 替换为对应的新增页面。

```
// snippet  in AmigoCallback
 private boolean handleLaunchActivity(Message msg) {
        ... ...
            ClassLoader classLoader = context.getClassLoader();
            Intent intent = (Intent) FieldUtils.readField(msg.obj, "intent");
            intent.setExtrasClassLoader(classLoader);
            Intent targetIntent = intent.getParcelableExtra(EXTRA_TARGET_
                                        INTENT);
            if (targetIntent != null) {
                ComponentName targetComponentName = targetIntent.
                                resolveActivity(context.getPackageManager());
                ActivityInfo targetActivityInfo =
getActivityInfoInNewApp(context, targetComponentName.getClassName());
                if (targetActivityInfo != null) {
                    targetIntent.setExtrasClassLoader(classLoader);
                    targetIntent.putExtra(EXTRA_TARGET_INFO,
                                          targetActivityInfo);
                    FieldUtils.writeDeclaredField(msg.obj, "intent",
                                                  targetIntent);
                    FieldUtils.writeDeclaredField(msg.obj, "activityInfo",
                                                  targetActivityInfo);
                }
            }
            ... ...
```

第三步，新增 Activity 即 StubActivity 的身份与 AMS 和 WMS 进行通信。

6.3.8 修复 so

修复 so 的思路，一般是通过反射操作 DexPathList 中 dexElements/nativeLibraryDirectories 数组属性，把新 so 的文件目录插到数组前面去。Amigo 处理有点不同，同上一步替换 ClassLoader 一样，在构建 ClassLoader 的时候直接把所有 so 的目录传参，摒弃了系统释放 so 的目录，这样 System.loadLibrary(String libName) 的时候会去自动查找 patch 的 so。

仅供非商业用途或交流学习使用

```
// snippet in AmigoClassLoader
public static AmigoClassLoader newInstance(Context context, String checksum)
{
        ... ...
        String libPath = getLibraryPath(context, checksum);
        ... ...
      return new AmigoClassLoader(apkPath, dexPath, dexOptDir, libPath,
                                  parent);
}
```

6.3.9 修复资源

修复资源比较简单，针对大多数 App，全局其实只有一个 AssetManager，以最新 APK 路径构造全新的 AssetManager，然后替换全局的 AssetManager 即可。

```
// snippet in PatchResourceLoader
static void loadPatchResources(Context context, String checksum) throws
Exception {
        AssetManager newAssetManager = AssetManager.class.newInstance();
        String path = PatchApks.getInstance(context).patchPath(checksum);
        invokeMethod(newAssetManager, "addAssetPath", path);
        invokeMethod(newAssetManager, "ensureStringBlocks");
        replaceAssetManager(context, newAssetManager);
}
```

6.3.10 差分实现

Amigo 接收的参数是一个 APK，但是现实情况下，APK 体积可能较大，这种情况下如果全量下发，则会存在下发成功率的问题。因此这个处理权 Amigo 是交出去的，由开发者自己去实现各自的差分算法。你可以选择基于文件整体差分的 bspatch，也可以选择 Amigo 提供的基于 APK 内部细粒度的 ZipPatch，也可以选择 Google play 使用的 archive-patcher，也可能是你自己实现的差分算法。上述几种差分实现均开源在 GitHub 上。

仅供非商业用途或交流学习使用

6.3.11 ROM 兼容性

Amigo 兼容范围较广，从 Android 2.1 到 Android 8.1。这里要归功于更少的反射、更少的 hook。

6.3.12 Amigo 技术局限

Amigo 有一定的技术局限：

- Amigo 面向的是单一 ClassLoader、单一 AssetManager 的 App，针对阿里开源的 atlas 这种插件化开发模式下的 App 可能不兼容，需要自定制。
- notification & widget 中 RemoteViews 的自定义布局不支持修改，只支持内容修复。

任何在 RemoteViews 里面使用的资源 ID 都需要进行这样的包装。

```
// 原来的写法
int color = context.getResources().getColor(resId)

// 现在的写法
int realResId = RCompat.getHostIdentifier(context, resId)
int color = context.getResources().getColor(realResId)
```

6.4 腾讯系热修复方案简介

关于腾讯系的热修复方案，我们前面在讲解冷启动热修复方式时已经有大量讲解，本节主要进行简单地介绍和总结。腾讯系热修复三大方案分别是：QQ 空间超级补丁方案，手机 QQ 的 QFix 方案，微信 Tinker 方案。这三者都是以冷启动方式实现的 dex 文件提前插入和替换。

要想真正理解腾讯系的冷启动修复，关键还是需要了解 Dalvik 虚拟机中的 dvmResolveClass 函数。

仅供非商业用途或交流学习使用

dvmResolveClass 函数的主要功能是用来加载一个类之前解析这个类的详细信息。对于预先插入 dex 方案，很可能在这个函数中发生 Class resolved by unexpected DEX 的问题，也就是我们通常说的 pre-verify 异常。

简单概括一下 pre-verify 问题就是，如果一个类引用到的所有类都和这个类在同一个 dex 文件中（Android 系统类除外），这个类就会在 odex 的时候被打上 CLASS_ISPREVERIFIED 标识。在运行时，在加载某个类的时候，如果发现引用这个类的类被打上了 CLASS_ISPREVERIFIED 标识，而这俩类又不在同一个 dex 中，就会和之前 CLASS_ISPREVERIFIED 的含义冲突，从而抛出这个异常。

由于补丁类是在补丁 dex 中的，而旧类是在原先 APK 的 dex 中，因而就会触发这个问题。

```
ClassObject* dvmResolveClass(const ClassObject* referrer, u4 classIdx,
    bool fromUnverifiedConstant)
{
    DvmDex* pDvmDex = referrer->pDvmDex;
    ClassObject* resClass;
    const char* className;

    /*
     * Check the table first -- this gets called from the other "resolve"
     * methods.
     */
    resClass = dvmDexGetResolvedClass(pDvmDex, classIdx);
    if (resClass != NULL)
        return resClass;

    LOGVV("--- resolving class %u (referrer=%s cl=%p)",
        classIdx, referrer->descriptor, referrer->classLoader);

    /*
```


仅供非商业用途或交流学习使用

```
     * Class hasn't been loaded yet, or is in the process of being loaded
     * and initialized now.  Try to get a copy.  If we find one, put the
     * pointer in the DexTypeId.  There isn't a race condition here --
     * 32-bit writes are guaranteed atomic on all target platforms.  Worst
     * case we have two threads storing the same value.
     *
     * If this is an array class, we'll generate it here.
     */
    className = dexStringByTypeIdx(pDvmDex->pDexFile, classIdx);
    if (className[0] != '\0' && className[1] == '\0') {
        /* primitive type */
        resClass = dvmFindPrimitiveClass(className[0]);
    } else {
        resClass = dvmFindClassNoInit(className, referrer->classLoader);
    }

    if (resClass != NULL) {
        /*
         * If the referrer was pre-verified, the resolved class must come
         * from the same DEX or from a bootstrap class.  The pre-verifier
         * makes assumptions that could be invalidated by a wacky class
         * loader.  (See the notes at the top of oo/Class.c.)
         *
         * The verifier does *not* fail a class for using a const-class
         * or instance-of instruction referring to an unresolveable class,
         * because the result of the instruction is simply a Class object
         * or boolean -- there's no need to resolve the class object during
         * verification.  Instance field and virtual method accesses can
         * break dangerously if we get the wrong class, but const-class and
         * instance-of are only interesting at execution time.  So, if we
         * we got here as part of executing one of the "unverified class"
         * instructions, we skip the additional check.
```


仅供非商业用途或交流学习使用

```
     *
     * Ditto for class references from annotations and exception
     * handler lists.
     */
    if (!fromUnverifiedConstant &&
        IS_CLASS_FLAG_SET(referrer, CLASS_ISPREVERIFIED))
    {
        ClassObject* resClassCheck = resClass;
        if (dvmIsArrayClass(resClassCheck))
            resClassCheck = resClassCheck->elementClass;

        if (referrer->pDvmDex != resClassCheck->pDvmDex &&
            resClassCheck->classLoader != NULL)
        {
            ALOGW("Class resolved by unexpected DEX:"
                 " %s(%p):%p ref [%s] %s(%p):%p",
                referrer->descriptor, referrer->classLoader,
                referrer->pDvmDex,
                resClass->descriptor, resClassCheck->descriptor,
                resClassCheck->classLoader, resClassCheck->pDvmDex);
            ALOGW("(%s had used a different %s during pre-verification)",
                referrer->descriptor, resClass->descriptor);
            dvmThrowIllegalAccessError(
                "Class ref in pre-verified class resolved to unexpected "
                "implementation");
            return NULL;
        }
    }

    LOGVV("##### +ResolveClass(%s): referrer=%s dex=%p ldr=%p ref=%d",
        resClass->descriptor, referrer->descriptor, referrer->pDvmDex,
        referrer->classLoader, classIdx);
```


仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        /*
         * Add what we found to the list so we can skip the class search
         * next time through.
         *
         * TODO: should we be doing this when fromUnverifiedConstant==true?
         * (see comments at top of oo/Class.c)
         */
        dvmDexSetResolvedClass(pDvmDex, classIdx, resClass);
    } else {
        /* not found, exception should be raised */
        LOGVV("Class not found: %s",
            dexStringByTypeIdx(pDvmDex->pDexFile, classIdx));
        assert(dvmCheckException(dvmThreadSelf()));
    }

    return resClass;
}
```

QQ 空间超级补丁的思路是，把这个 CLASS_ISPREVERIFIED 标识去除，也就是破坏打这个标识的条件，让每个类都必然引用一个不在本 dex 中的类，当然，如果执行到这个引用处，就会触发找不到类的异常。所以，在插入引用代码后，还需要在运行的时候，不走到这个引用处，这样就可以避免崩溃。

这个方案的问题是，需要对每个类插入额外代码，本身会使得代码臃肿，并且由于没有 CLASS_ISPREVERIFIED 标识，会使得 Resolve 类的时间变长，严重影响性能。

QFix 的思路是，先执行 dvmResolveClass 函数，并且传入参数 fromUnverifiedConstant 为 false，这样就可以跳过 pre-verify 检查的逻辑判断，直接执行 dvmDexSetResolvedClass，后面再次进入 dvmResolveClass 就不会引发问题。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

不过这个方案有个致命的问题，就是在原有已存在的类中新增 public/protect/default 方法，条件被严重局限。这个在之前 3.2 节多态对冷启动的影响中已经有详细说明，这里不再赘述。

Tinker 采用的方案是基于补丁和原 APK 中的 dex 包，完整合成全量 dex，并且抛弃原生机制的 dex 和采用这个新合成的 dex，这就可以避免补丁类和旧类所属 DEX 不一致的问题。从后续的技术发展来看，Tinker 也是目前腾讯系热修复中最为稳定可靠的方案。

另外，Tinker 的一大亮点是，极致的 dex 补丁差量操作。它们生成补丁包时比较粒度十分精细，实现了代码级别的差异比较，而不是传统的 BsDiff 算法。合成的时候也是根据补丁和原有 APK 的 dex 进行精细的合成操作。这一套差异合成机制确实是需要花费大量时间研究的，开发者的工匠精神值得赞赏。

不过，精细合成的问题就在于性能损耗，由于粒度太细，不仅生成补丁要花费比较多的时间，手机端上的合成操作更是相当消耗性能。

在 ART 运行时环境下，不存在 pre-verify 问题。直接插入补丁 dex 是可以正常修复的，不过在 7.0 之后 JIT 机制，使得只有完整 dex 方案的 Tinker 存活了下来，超级补丁和 QFix 无法解决 Android JIT 机制导致的旧类的热代码残留问题——加载一个类的时候始终会解析到原有 APK 的 dex 中被 JIT 的旧类，而无法解析到补丁类。

Tinker 方案目前作为冷启动方案的代表，已经有很多开发者采用，但是其接入成本比较高，不仅需要改造编译环境，各种复杂的配置以及 Application 的改造也令人头疼。并且随着 Gradle 的升级，Tinker 必须不断适配兼容新版本 Gradle，开发者的维护成本也是比较大的。

不过，作为早期冷启动修复技术的主要代表，腾讯系的热修复方案给业界带来了新的思路，在早期 Android 热修复的探索中，使广大开发者看到了通过研读系统代码，找到突破点进行改造，从而改变代码执行逻辑的希望。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 7 章
热修复技术的未来展望

对于热修复技术未来的畅想与期盼。

仅供非商业用途或交流学习使用

7.1 热修复的专业性

自 Sophix 热修复技术推出以来，已经有数千 App 踊跃接入，网络上各种接入教程和文章不断发布，目前阿里云平台统计到的使用设备数已过千万。从中可见，开发者对热修复技术的需求是如此急迫。根本原因是，不论是什么类型的 App，对更新和及时修复的诉求都是一样的，只是依赖的程度不同。从 App 的发展阶段来看，对于成熟期的 App，用户量大，需要功能快速上线到尽可能多的设备上，避免碎片化。而对于创业期的 App 而言，在快速竞争的环境下，代码有时难免粗糙，这时也急需具有问题出现后的止损能力。而对于更多的 App 而言，功能更新和问题修复的需求是同时存在的，而热修复技术就是为此而生的。

热修复是一个与业务完全无关的模块，开发者如果要自己实现一套可靠的热修复框架，将花费大量时间和精力。虽然市场上已经有很多开源的热修复方案，但是其中的很多陷阱，往往要踩过才知道，等你把这些陷阱一一踩过之后，可能大量的用户已经对你失去信心。并且，采用开源的热修复方案，看似免费，但并不是代表没有成本，最直观的就是服务器成本，还有开发人员的投入成本。开源的热修复方案，离真正部署上线可用，还有很长的路要走。热修复并不是简单的客户端 SDK，它还包含了安全机制和服务端的控制逻辑，这整条链路也不是短时间内可以快速完成的。建设一个功能完备稳定性好的热修复体系，并非朝夕之功。

所以，依靠一个稳定可靠、简单实用的商业版本，反而能使各方面的成本降到最低。还是那句老话，专业是事交给专业的人去做，开发者应该把更多时间精力放到自

仅供非商业用途或交流学习使用

己的核心业务之中。尤其对于热修复这样一个系统底层依赖极强的领域，一个小的疏忽可能就会导致大规模的崩溃，想要真正把控好热修复方案，往往需要很专业的工程师来不断解决问题。即使市场上有免费的热修复服务可以用，我相信，选择它们绝不是一个成熟企业的明智之举，免费的服务，它的质量如何？能不能及时帮助开发者解决问题？对于非常急迫的问题，如果解决不了，能不能有专人来协助处理？它将来如果收费，应该如何抉择？一个优秀的技术服务，若想实现良性循环，不断改进产品质量，付费是必由之路。只要价格合理，就可以实现 App 用户、App 开发者、服务提供方的三者共赢。

Sophix 提供了一套更完美的客户端到服务端一体的热更新方案。做到了图形界面一键打包、加密传输、签名校验和服务端控制发布与灰度功能，让你用最少的时间实现最强大可靠的全方位热更新。并且在代码修复、资源修复、SO 库修复方面，都做到了业界最佳。而更为重要的是，Sophix 的接入是如此的简洁快速，因为我们在设计之初，就是从用户的使用角度考虑的，这里面兼顾了灵活性和优雅性，Sophix 的使用和接入成本是目前所有热修复方案中最低的，这一点也可以从网上众多接入文档的叙述中得到印证。

目前对于月活 5 万以下的 App，我们是不进行收费的，这也是为了改善 Android 生态，帮助小型初创期的 App，让它们也能够享受到可靠的热修复服务。我们十分重视开发者的使用反馈，所有反馈给我们的问题和建议，都是以最快的速度响应并彻底解决的，Sophix 正是如此在市场的检验下稳健成长并成熟。

7.2 对 Android 生态的影响

很多人会把热修复技术跟其他国内厂商的“黑科技”混为一谈。有人说，你们国内开发者就是瞎搞，就不能给我们 Android 用户一个更加纯净的环境吗？

这里笔者需要澄清一下。热修复技术不同于其他国内的 Android “黑科技”。就

仅供非商业用途或交流学习使用

比如，国内 Android 进程保活，是让 App 持续驻留在后台避免被系统杀死，这既耗费手机电量又占内存，浪费了很多手机资源。再比如，App 自行定制的推送服务，无节操地对用户进行信息轰炸。还有更过分的全家桶，一个 App 同时拉起一堆 App，并且长期占着内存，使得手机卡顿不堪。总归，这些技术都是为了 App 厂商的利益而损害手机使用者的实际体验。

而热修复技术是完全不同的，它达到的是一个手机用户和开发者双赢的目的。不仅厂商可以快速迭代更新 App，使得功能可以最快上线。并且由于热更新过程是毫无感知的，手机用户也减少了烦琐的更新步骤，节省了大量等待更新的时间。这实际上是改善了 Android 的生态环境。只是这其中最重要的，是要保证热修复功能的稳定性。而 Sophix 的稳定性，是经过了无数开发者检验的，并且还有手淘多年深厚的技术沉淀作为保障。

7.3 Android 与 iOS 热修复的不同

前段时间，苹果公司封杀了 iOS 的热修复功能，这给 iOS 的开发者带来了很大困扰。

热修复功能被禁止，会使得很多 App 不得不靠直接发版进行更新，这样一旦新版本出了问题，整个更新迭代过程变得十分漫长。并且一些试验性功能无法进行灰度测试，这就使得一个重要功能的更新将直接全量发版，如果功能不够稳定，波及范围就变得非常广。而且，用户需要重新下载整个 App，不仅流程漫长，原本不到 1MB 的补丁就能解决的事，现在不得不下载几十甚至上百 MB 的完整包才能更新。

苹果公司这一政策的推出，使得很多人也因此不看好 Android 的热修复技术了。在这里，我们可以打消这种错误的观念。因为 Android 的情况和 iOS 是有极大不同的。主要有两个方面：

仅供非商业用途或交流学习使用

- 谷歌和苹果在中国的地位不同；
- Android 和 iOS 的开放性不同。

首先，Google 向来是一家倡导开放自由的公司，在技术上很少会采用随意封杀的政策。前段时间虽然开始禁用私有 API，但是还是会提供灰名单来放松一些限制，我们上报的一些热修复中使用的私有 API 也都很顺利地被采纳进灰名单中，从中可见 Google 对开发者声音非常重视并具有较好的技术包容心。并且从技术上来看，由于处于自身进程，理论上总是有办法突破这层限制，目前网上也涌现了很多对私有 API 调用封禁的绕过手段。

Google 在中国没有像苹果公司那样的控制力，即使它想要封杀也不可能，国内是有各个安卓应用市场的，没有统一的 App 安装渠道。另外，Android 是开源的，各个厂商都可以做定制，并且各家都有自己的应用商店，不存在和 iOS 上的 AppStore 一样的唯一的受管控的应用安装渠道。

7.4 未来，无限可能

我们对于未来是很乐观的，Android 的热修复领域不仅不会受到封杀，反而还有很大的发展空间。目前阿里聚安全加固已经支持了 Sophix，我们也与其他主流加固厂商保持长期联系，目前也都基本做到了兼容。热修复结合安全加固，将会使得 App 的稳定性和安全性更加坚不可摧。甚至后续还可以与系统厂商合作，对系统 App 乃至系统组件进行修复，这样就可以避免频繁 OTA 升级。

因此，热修复所能发挥的价值将是十分巨大的。热修复还可以与其他领域进行碰撞，引发无限的可能性。在这里，我们非常期待携手广大应用厂商以及 ROM 厂商，共同推动 Android 的生态更加完善。

附录 A
Sophix 方案比较

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

A.1 Sophix 方案纵向比较

方案对比	Andfix开源版本	阿里Hotfix 1.X	阿里Hotfix 最新版
方法替换	支持，除部分情况 [0]	支持，除部分情况	**全部支持**
方法增加减少	不支持	不支持	以冷启动方式支持 [1]
方法反射调用	只支持静态方法	只支持静态方法	以冷启动方式支持
即时生效	**支持**	**支持**	**视情况支持** [2]
多 DEX	不支持	**支持**	**支持**
资源更新	不支持	不支持	**支持**
so 库更新	不支持	不支持	**支持**
Android 版本	支持 2.3~7.0	支持 2.3~6.0	**全部支持包含 7.0 以上**
已有机型	大部分支持 [3]	大部分支持	**全部支持**
安全机制	无	**加密传输及签名校验**	**加密传输及签名校验**
性能损耗	**低，几乎无损耗**	**低，几乎无损耗**	**低，仅冷启动情况下有些损耗**
生成补丁	繁琐，命令行操作	繁琐，命令行操作	**便捷，图形化界面**
补丁大小	**不大，仅变动的类**	**小，仅变动的方法**	**不大，仅变动的资源和代码** [4]
服务端支持	无	**支持服务端控制** [5]	**支持服务端控制**

说明：

[0] 部分情况指的是构造方法、参数数目大于 8 或者参数包括 long,double,float 基本类型的方法。

[1] 冷启动方式，指的是需要重启 App 在下次启动时才能生效。

[2] 对于 Andfix 及 Hotfix 1.X 能够支持的代码变动情况，都能做到即时生效。而对于 Andfix 及 Hotfix 1.X 不支持的代码变动情况，会走冷启动方式，此时就无法做到即时生效。

[3] Hotfix 1.X 已经支持绝大部分主流手机，只是在 X86 设备，以及修改了虚拟机底层结构的 ROM 上不支持。

[4] 由于支持了资源和库，如果有这些方面的更新，就会导致的补丁变大一些，这个是很正常的，并且由于只包含差异的部分，所以补丁已经是最大程度的小了。

[5] 提供服务端的补丁发布和停发、版本控制和灰度功能，存储开发者上传的补丁包。

仅供非商业用途或交流学习使用

A.2 Sophix 方案横向比较

方案对比	Sophix	Tinker	Amigo
dex 修复	**同时支持即时生效修复和冷启动修复**	冷启动修复	冷启动修复
资源更新	**差量包，不用合成**	差量包，需要合成	全量包，不用合成
so 库更新	**插桩实现，开发透明**	替换接口，开发不透明	**插桩实现，开发透明**
性能损耗	**低，仅冷启动情况下有些损耗**	高，有合成操作	**低，全量替换**
生成补丁	**直接选择已经编好的新旧包在本地生成**	编译新包时设置基线包	上传完整新包到服务端
补丁大小	**小**	**小**	大
接入成本	**傻瓜式接入**	复杂	一般
Android 版本	**全部支持**	**全部支持**	**全部支持**
安全机制	**加密传输及签名校验**	**加密传输及签名校验**	**加密传输及签名校验**
服务端支持	**支持服务端控制**	**支持服务端控制**	**支持服务端控制**

欢迎您加入读者群，您可以在群里提任何关于本书的建议和意见。我们也会在该群中发布本书的所有动态和新闻。

关注微信，
发送34389加入本书共读群

移动热修复
产品使用交流钉钉群

Broadview®
www.broadview.com.cn

专家力荐

自2014年至今，手机淘宝引领了业界Android系统组件化和热修复技术的风潮，后来者Instant App或多或少也受到了国内技术的影响。今天看到团队成员将热修复技术认真系统地整理成书，非常欣喜。在这本书中，既能看到对热修复技术发展历史系统深入的总结，又能看到国内程序员在Android系统级技术持续突破上做出的不懈努力，更可以看到国内程序员坚持打造优秀专业移动技术产品的雄心壮志！

吴志华（天施）
手机淘宝基础平台部负责人，阿里巴巴资深技术专家

业内少有的深度讲解Android系统热修复技术的书籍，对于原理、代码讲解得非常清晰和深入，值得Android工程师研读。

倪生华（玄黎）
手机淘宝基础架构团队负责人，阿里巴巴资深技术专家

应用热修复是一项略带神秘而又颇具争议的技术，但是它的确赋予了应用开发者“驾着飞机修引擎”的能力。本书从Android系统应用热修复技术的原理及代码实现、多种方案进行比较的角度，系统地阐述了Android平台的应用热修复技术。对Android系统应用热修复技术有好奇心的技术人员，这本专题书不容错过。

潘爱民
计算机技术领域作家，阿里巴巴飞猪事业部首席架构师

2015年阿里无线在业界首次推出Android热修复技术Dexposed，该技术为Android底层技术服务于业务痛点需求指明了一个崭新的方向，掀起了业界百花齐放的探索热潮。Sophix的发布让我们再次看到了阿里无线在这个技术领域的自我迭代和锐意创新。这是一个技术改变格局的时代，同时也是一个能人辈出的时代！

冯森林
安卓绿色联盟发起人，手机淘宝前架构师

上架建议：移动开发>Androi

ISBN 978-7-121-34389-6

9 787121 343896

定价：79.00元

策划编辑：孙学瑛
责任编辑：孙学瑛
封面设计：李　玲

关注阿里技术
把握前沿技术脉搏

欢迎了解阿里
EMAS移动热修复产品